VISIT…

Mastering XML Transformations

2nd Edition
Now covers XSLT 2.0

XSLT

O'REILLY®

Doug Tidwell

O'REILLY®

XSLT

XML documents are especially useful when they're transformed into familiar formats, such as web pages, PDF files, or Java code—and those conversions require mastery of XSLT. This practical book not only teaches you how to be productive using XSLT, it also serves as a handy, dictionary-style reference to all the features and functions of the language that you'll need on the job. With *XSLT*, Second Edition, you will:

- Learn XSLT basics, including simple stylesheets and methods for setting up transformation engines
- Walk through the many parts of XSLT, particularly the template-based approach to transformations
- Understand the basics of XPath versions 1.0 and 2.0—the language used to describe parts of an XML document
- See how XML Schema support works in XSLT 2.0, including how to define elements and datatypes and use them in your stylesheets
- Get hundreds of stylesheets, including examples for every element, function, and operator defined by XSLT and XPath
- Use examples that apply both XSLT 1.0 and 2.0 solutions to the same problems so you can decide which version is more appropriate for your project

This new edition includes a complete set of stylesheet examples for both XSLT 1.0 and 2.0. You also get a thorough explanation of the relationship between XSLT, XPath, and other web standards. If you use XSLT, this book is the one resource you'll want on hand to help you solve problems quickly and accurately.

"The best review I received for the first edition of this book began, 'I will never read this book.' This was actually a positive review, as the reviewer went on to explain. 'When I have a problem, I grab this book off the shelf, go to the index, and within five minutes I've found the answer to my problem. Then I toss it back on the shelf.'"

—Doug Tidwell,
from the Preface of this book

Doug Tidwell, a senior programmer at IBM, has been working with markup languages for more than two decades. An expert who has been involved with XML since he spoke at the first SGML/XML conference in 1997, he teaches XML classes around the world.

www.oreilly.com

US $49.99 **CAN $49.99**

ISBN: 978-0-596-52721-1

9 780596 527211 54999

Safari Books Online

Free online edition for 45 days with purchase of this book. Details on last page.

XSLT

Other resources from O'Reilly

Related titles

XSLT Cookbook™	XML Hacks™
XQuery	XSLT 1.0 Pocket Reference
Learning XSLT	Relax NG
Java & XML	Unicode Explained
Schematron	XML in a Nutshell
Developing Feeds with RSS and Atom	Learning XML

oreilly.com

oreilly.com is more than a complete catalog of O'Reilly books. You'll also find links to news, events, articles, weblogs, sample chapters, and code examples.

oreillynet.com is the essential portal for developers interested in open and emerging technologies, including new platforms, programming languages, and operating systems.

Conferences

O'Reilly Media, Inc. brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today for free.

SECOND EDITION

XSLT

Doug Tidwell

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

XSLT, Second Edition
by Doug Tidwell

Copyright © 2008 O'Reilly Media, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safari.oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Simon St.Laurent
Production Editor: Sarah Schneider
Proofreader: Mary Brady
Indexer: Fred Brown
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
June 2008: Second Edition.
August 2001: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *XSLT*, the image of a Jabiru, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations uses by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-0-596-52721-1

[C]

1213384691

To my family—my wonderful wife, Sheri Castle, and our amazing daughter, Lily—for their love, support, and understanding. Nothing I do would be possible or meaningful without them.

...and a special thanks to our dog, Domino, who frequently and selflessly pushed his fuzzy head between my hands and keyboard to protect me from carpal tunnel syndrome. Good boy!

Table of Contents

Preface ..... **xi**

1. Getting Started **1**
The Design of XSLT 1
XML Basics 4
Installing XSLT Processors 20
Summary 24

2. The Obligatory Hello World Example **25**
Goals of This Chapter 25
Transforming Hello World 25
How a Stylesheet Is Processed 27
Stylesheet Structure 30
Sample Gallery 36
Summary 44

3. XPath: A Syntax for Describing Needles and Haystacks **45**
The XPath Data Model 46
Location Paths 55
Attribute Value Templates 66
Datatypes 67
XPath Operators 71
[2.0] Comments in XPath Expressions 102
[2.0] Types of XSLT 2.0 Processors 104
The XPath View of an XML Document 104
Summary 112

4. Creating Output **113**
Goals of This Chapter 113
Generating Text 113
Numbering Things 118
Formatting Decimal Numbers 127

[2.0] Formatting Dates and Times 130
Using <xsl:copy> and <xsl:copy-of> 132
Dealing with Whitespace 139
Summary 144

5. Branching and Control Elements 145
Goals of This Chapter 145
Branching Elements of XSLT 145
Invoking Templates by Name 151
Parameters 152
Variables 167
Using Recursion to Do Most Anything 169
A Stylesheet That Emulates a for Loop 174
Summary 179

6. Creating Links and Cross-References 181
Using the XML ID, IDREF, and IDREFS Datatypes 181
XSLT's Key Facility 194
Generating Links in Unstructured Documents 198
Summary 204

7. Sorting and Grouping Elements 205
Sorting Data with <xsl:sort> 205
[2.0] The <xsl:perform-sort> Element 215
Grouping Nodes 219
[2.0] New Grouping Syntax in XSLT 2.0 228
Summary 243

8. Combining Documents 245
The document() Function 245
The document() Function and Sorting 254
Implementing Lookup Tables 254
Grouping Across Multiple Documents 257
[2.0] Using XSLT 2.0 to Simplify Things 260
[2.0] The doc() and doc-available() Functions 269
[2.0] The collection() Function 271
[2.0] The unparsed-text() and unparsed-text-available() Functions 272
Summary 275

9. Extending XSLT 277
The XSLT Extension Mechanism 277
[2.0] Creating New Functions with <xsl:function> 279
Example: Generating Multiple Output Files 281

Creating Custom Collations 287
Generating Hidden Word Graphics 293
Example: Generating an SVG Pie Chart 303
Writing Extensions in Other Languages 326
Using Extension Functions from the EXSLT Library 330
Accessing a Database with an Extension Element 333
Creating a Photo Album with an Extension Element 339
Summary 360

A. XSLT Reference 361

B. XPath Reference 545

C. XSLT, XPath, and XQuery Function Reference 563

D. XML Schema Overview 871

E. [2.0] Regular Expressions 897

F. XSLT Formatting Codes 919

G. XSLT 2.0 Migration Guide 925

Glossary 933

Index 943

Preface

About This Book

The goal of this book is to help you make the most of XSLT, the Extensible Stylesheet Language for Transformations. It covers both XSLT 1.0 and XSLT 2.0, along with versions 1.0 and 2.0 of XPath, the XML Path Language. The two languages are designed to work together: XPath identifies the parts of an XML document that should be transformed, and XSLT says how the transformation should be done.

The first few chapters of the book cover the features of XSLT by solving common problems using the language. Once you've mastered those techniques, the last section of the book contains a complete set of examples for all the features of XSLT and XPath. The book is designed as a tutorial for learning the language as you're getting started. Once you're comfortable with XSLT, the book can be used as a dictionary-style reference for the features and functions of the language.

Where I'm Coming From

Before we begin, it's only fair that I tell you my biases.

I Believe in Open, Platform-Neutral, Standards-Based Computing

If any part of your business life ties you down to anything closed, proprietary, or platform-specific, I encourage you to make some changes. This book shows you how to take charge of your data and move it from one place to another on your terms, and not your software vendor's. XML is shifting the balance of power from vendors to software users. If your tools force you to work in unnatural ways or refuse to let you have your data when and where you want it, you don't have to take it anymore.

I Assume You're Busy

The best review I received for the first edition of this book began, "I will never read this book." This was actually a positive review, as the reviewer went on to explain. "When

I have a problem, I grab this book off the shelf, go to the index, and within five minutes I've found the answer to my problem. Then I toss it back on the shelf."

That's exactly the kind of book I've tried to write. There are hundreds of stylesheets in this book, including examples for every XSLT element, function, and operator defined by XSLT and XPath. The first chapters of the book are prose that explain how stylesheets work and what you need to learn to be productive with XSLT. Once you're comfortable with that material, you can use the rest of the book as a dictionary-style reference.

I Don't Care Which Standards-Compliant Tools You Use

My job as an author and a teacher is to show you how to use standards-compliant tools to simplify your life. I'm not here to sell you a parser, an XSLT processor, a toaster, or anything else, so please use whatever tools you like. I encourage you to take a look at all of the tools out there and find your own preferences. As I wrote this edition of the book, I used four processors to test the examples:

- Almost all of the examples were tested with Michael Kay's excellent Saxon XSLT processor. The open source edition of Saxon supports all of the XSLT 2.0, XPath 2.0 and XQuery 1.0 specs *except* for the schema-specific functions. As the editor of the XSLT 2.0 specification, Dr. Kay's processor is currently the most complete implementation of XSLT 2.0.

  Saxon-B (the basic processor without schema support) is available here: *http://saxon.sourceforge.net/*. The SourceForge project page is at *http://sourceforge.net/projects/saxon*. Saxon is available in Java and .NET versions.

  There is also a commercial version of Saxon that includes full schema support. For more information on Saxon-SA, which is the schema-aware version, visit *http://www.saxonica.com/*.
- The XSLT engine from Altova XML Spy was also used for all of the XSLT 2.0 examples. The Altova XSLT engine, although not open source, does provide complete schema support in a no-cost product. The license for the Altova engine currently allows you to redistribute it with your own code. To get the engine and the license terms, visit *http://www.altova.com/altovaxml.html*.
- Apache's Xalan XSLT engine supports almost all of the XSLT 1.0 examples in the book. (The XSLT 1.0 stylesheets that it doesn't support are ones that use extensions written for other processors.) It's also a forwards-compatible XSLT processor, so it can work with XSLT 2.0 stylesheets.

  The Java version of the processor, Xalan-J, is available at *http://xml.apache.org/xalan-j/*. There's also a C++ version at *http://xml.apache.org/xalan-c/*.
- Microsoft's .NET framework supports XSLT 1.0, as does the MSXSL utility. One significant addition to this edition is more focus on the Microsoft platform. In

addition to testing all of the XSLT 1.0 samples with the Microsoft tools, there are also XSLT extensions written in C# and EcmaScript.

The MSXSL XSLT processor is available from the Microsoft XML downloads page, *http://msdn.microsoft.com/XML/XMLDownloads/default.aspx*. There is also an XSLT processor embedded in the .NET framework; it's part of the `System.Xml.Xsl` namespace.

XSLT Is a Tool, Not a Religion

An old adage says that to a person with a hammer, everything looks like a nail. I don't claim that XSLT is the solution to every business problem you'll encounter. Chapter 1 discusses reasons why XML and XSLT were created and the design decisions behind XSLT, and it tries to identify the kinds of problems XSLT is designed to solve. All chapters in this book illustrate common scenarios in which XSLT is extremely powerful and useful.

That being said, if a particular tool does something better than XSLT does, then by all means, use that other tool. For example, XSLT has functions for sorting and grouping. If the data you're transforming comes from a relational database, it's probably far more efficient to use the `ORDER BY` and `GROUP BY` features of your database instead of sorting and grouping with XSLT. XSLT is a powerful addition to your tool box, but that doesn't mean you should throw out all your other tools.

You Shouldn't Migrate All of Your Stylesheets Just Because There's a New Version of XSLT

Anytime a new version of a language, standard, or software package comes along, deciding when or if to migrate to the new features depends on your application. If you've built a web application in which you use a web browser to process XSLT stylesheets on the client side, you can't migrate to XSLT 2.0 until all the major browsers support XSLT 2.0. That's going to be a while. On the other hand, if you use XSLT to transform your data and then send the transformed data to the client, you can use XSLT 2.0 right away. With very few exceptions, anything that worked in XSLT 1.0 works in XSLT 2.0. We cover migration in Appendix G.

XSLT 2.0 and XPath 2.0 have many new features that make your stylesheets easier to write, easier to maintain, and much more powerful. It's definitely worth your time to investigate the new features to see how many of them you can use.

How This Book Is Organized

XSLT 2.0 has added significant new features to the language, many of which are related to the changes in XPath 2.0. The biggest challenge I had as an author was figuring out how to organize the book. One approach would have been to make this an XSLT 2.0

book, writing under the assumption that everyone would migrate to XSLT 2.0 as soon as possible. I don't believe that will happen, so I didn't go that way. Instead, I tried to cover everything in terms of common tasks, things you'll probably have to do with XSLT. If there are new features in XSLT 2.0 that apply to those tasks, I mention them after explaining the concepts behind the stylesheets. Usually XSLT 2.0 makes your life much easier, so I begin the discussion by pointing out that if you're using XSLT 2.0, you've got a simpler option.

As with the first edition, this book has two parts: a series of prose chapters that cover concepts and tasks, followed by a series of appendixes that form a reference to all of the elements, functions, operators, and other details you'll need as you write stylesheets. Once you're comfortable with XSLT, you can use the appendixes as a dictionary of all things related to XSLT and XPath.

The book contains the following chapters:

Chapter 1, *Getting Started*
: Covers the basics of XML and discusses how to install the stylesheet engines used in this book.

Chapter 2, *The Obligatory Hello World Example*
: Takes a look at an XML-tagged "Hello World" document, then examines stylesheets that transform it into other things.

Chapter 3, *XPath: A Syntax for Describing Needles and Haystacks*
: Covers the basics of XPath, the language used to describe parts of an XML document. This chapter includes an in-depth discussion of the many changes introduced in XPath 2.0.

Chapter 4, *Creating Output*
: Discusses the basics of creating output, including extracting text, copying information, and numbering things.

Chapter 5, *Branching and Control Elements*
: Discusses the logic elements of XSLT (`<xsl:if>` and `<xsl:choose>`) and how they work. Also covers the new `if` operator in XPath 2.0.

Chapter 6, *Creating Links and Cross-References*
: Covers the different ways to build links between elements in XML documents. Using XPath to describe relationships between related elements is also covered.

Chapter 7, *Sorting and Grouping Elements*
: Goes over the `<xsl:sort>` element and discusses various ways to sort elements in an XML document. It also talks about how to do grouping with various XSLT elements and functions. Grouping is much simpler in XSLT 2.0; the new grouping features are covered in this chapter as well.

Chapter 8, *Combining Documents*
: Discusses the `document( )` function, which allows you to combine several XML documents, then write a stylesheet that works against the collection of documents. Related functions from XSLT 2.0 are also featured.

Chapter 9, *Extending XSLT*
: Explains how to write extension elements and extension functions. Although XSLT and XPath are extremely powerful and flexible, there are still times when you need to do something that isn't provided by the language itself.

The last section of the book contains reference information:

Appendix A
: An alphabetical listing of all the elements defined by XSLT, with examples for those elements and how they were designed to be used.

Appendix B
: A listing of various aspects of XPath, including datatypes, axes, node types, and operators.

Appendix C
: An alphabetical listing of all the functions defined by XPath and XSLT.

Appendix D
: Provides a brief overview of XML Schema. One of the additions to XSLT 2.0 is the ability to use XML Schemas to define datatypes and validate XML structures against them.

Appendix E
: Covers the syntax and features of the regular expression language used by XPath 2.0 and XSLT 2.0.

Appendix F
: Provides a handy listing of all the formatting codes used in XSLT and XPath.

Appendix G
: Lists a number of considerations and approaches for migrating to XSLT 2.0.

Glossary
: A glossary of terms used in XSLT, XPath, and XML in general.

Conventions Used in This Book

Items appearing in this book are sometimes given a special appearance to set them apart from the regular text. Here's how they look:

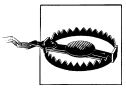
Italic
: Used for citations of books and articles, commands, email addresses, introduction of terms, and URLs

`Constant width`
: Used for literals, constant values, code listings, and XML markup

`Constant-width bold`
: Used to indicate user input

`Constant-width italic`
: Used for replaceable parameter and variable names

This icon represents a tip, suggestion, or general note.



This icon represents a warning or caution.

[1.0]
: This text represents information that applies *only* to XSLT 1.0 and XPath 1.0.

[2.0]
: This text represents information that is new in XSLT 2.0 and XPath 2.0.

[2.0 – Schema]
: This text represents information that applies to schema-aware XSLT 2.0 processors.

How to Contact Us

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

To ask technical questions or comment on the book, send email to:

bookquestions@oreilly.com

The web site for this book lists examples, errata, and plans for future editions. You can access this page at:

http://www.oreilly.com/catalog/9780596527211

For more information about our books, conferences, software, resource centers, and the O'Reilly Network, see our web site:

http://www.oreilly.com

Safari® Enabled

When you see a Safari® Enabled icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*.

Acknowledgments for the Second Edition

I want to thank Jeni Tennison for being the lead reviewer of this edition. Her ability to see through to the essence of a problem and point out the simplest and most elegant way to solve it is astounding. I have blisters from smacking my forehead as I read her review comments, thinking at the time, "Of course! I should have seen that right away." Jeni, thank you.

I also benefited from Patricia Walmsley's excellent review, especially in the appendixes that cover all the elements and functions in XSLT, XPath, and XQuery. The examples and terminology in those sections are far more useful and correct as a result.

A big thanks to Michael Kay for providing a copy of Saxon-SA to test the schema examples in the book. The entire XSLT community owes him an enormous debt for making the XSLT 2.0 spec robust, readable, and complete, and for writing the Saxon XSLT engine.

This book was written entirely in DocBook, a very powerful XML vocabulary for publishing. Two books have been invaluable as I've worked with DocBook. The first is O'Reilly's *DocBook: The Definitive Guide*, written by Norm Walsh and Leonard Mueller (available online at *http://www.oreilly.com/catalog/docbook/chapter/book/docbook.html*). If you want to know anything about DocBook, this is the place to look. The open source community also maintains an extremely sophisticated set of XSLT stylesheets that transform DocBook into a variety of other formats. For help in using the DocBook XSL, Bob Stayton's *DocBook XSL: The Complete Guide* (Sagehill Enterprises; available online at *http://sagehill.net/book-description.html*) was invaluable. Thanks to all three of these great authors.

I also want to thank the people I've worked with over the last few years. The IBM developerWorks team is still a great influence on me. I'll always think of myself as part of the developerWorks family. During my time with IBM's Developer Skills organization, I had the great pleasure of working with an incredibly talented team. That group is paid to give away as much knowledge as possible, along with free software to professors and students around the world. Finally, I want to thank the members of my current team in IBM's Software Group Strategy organization. I'm very happy to be working again for Dirk Nicol, the father of developerWorks.

I will resist the temptation to name names here in fear of forgetting someone. I hope all of you know how much you mean to me, and how much I've learned from all of you.

Finally, I want to thank Simon St.Laurent for his guidance on the second edition. Both of us were nervous about figuring out how to add XSLT 2.0 and XPath 2.0 to this book without creating a 5,000 page tome. Unfortunately, I also relied on Simon's patience as portions of the book took far longer than either of us had hoped. Simon, you're the best.

Acknowledgments from the First Edition

First and foremost, I'd like to thank the reviewers of this book. David Marston of Lotus was the lead reviewer; David, thank you so much for your comments, wisdom, and knowledge. Along the way, I also got a lot of good feedback and encouragement from Tony Colle, Slavko Malesvic, Dr. Joe Molitoris, Shane O'Donnell, Andy Piper, Sreenivas Ramarao, Mike Riley, and Willie Wheeler. This book is significantly better because of your comments and other efforts.

I'd also like to thank my teammates at developerWorks for encouraging me to undertake this project. Taking on an additional full-time job hasn't been easy, but their advice, flexibility, and understanding as I've tried to balance my responsibilities has been invaluable. Even more valuable is the fact that I'm surrounded by some of the most interesting, creative, and remarkable people I've ever known. You guys rule.

For the times I've been at home (in Raleigh, North Carolina), I've depended on my nutritional advisors at Schiano's Pizza: "Hey, you want your usual?" (Slight pause.) "Yeah, that'd be great, thanks." Nothing's as comforting as a couple of slices. If you're within a day's drive of Raleigh, I strongly encourage you to visit.

Finally, I'd like to thank the staff at O'Reilly, especially Laurie Petrycki and Simon St.Laurent. Laurie, thank you for convincing me to take on this project and for sticking with me when my ability to find the time to write was in doubt. Simon, I've enjoyed reading your books for years; it's been an honor to work with you. Your guidance, technical insight, patience, and suggestions were invaluable.

Thanks so much to all of you!

CHAPTER 1

Getting Started

In this chapter, we review the design rationale behind XSLT and XPath and discuss the basics of XML. We also talk about other web standards and how they relate to XSLT and XPath. We conclude the chapter with a brief discussion of how to set up an XSLT processor on your machine so you can work with the examples throughout the book.

The Design of XSLT

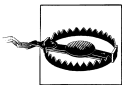
XML went from working group to entrenched buzzword in record time. Its flexibility as a language for presenting structured data made it the lingua franca for data interchange. Early adopters used programming interfaces such as the Document Object Model (DOM) and the Simple API for XML (SAX) to parse and process XML documents. As XML became mainstream, however, it was clear that the average web citizen couldn't be expected to hack Java, Visual Basic, Perl, or Python code to work with documents. What was needed was a flexible, powerful, yet relatively simple language capable of processing XML.

What the world needed was XSLT.

XSLT, the Extensible Stylesheet Language for Transformations, is an official recommendation of the World Wide Web Consortium (W3C). It provides a flexible, powerful language for transforming XML documents into something else, such as an HTML document, another XML document, a Portable Document Format (PDF) file, a Scalable Vector Graphics (SVG) file, a Virtual Reality Modeling Language (VRML) file, Java code, a flat text file, a JPEG file, or most anything you want. You write an XSLT stylesheet to define the rules for transforming an XML document, and the XSLT processor does the work.

The W3C has defined two families of standards for stylesheets. The oldest and simplest is Cascading Style Sheets (CSS), a mechanism used to define various properties of markup elements. Although CSS can be used with XML, it is most often used to style HTML documents. I can use CSS properties to define certain elements to be rendered in blue, or in 58-point type, or in boldface. That's all well and good, but there are many things that CSS can't do:

- CSS can't change the order in which elements appear in a document. If you want to sort certain elements or filter elements based on a certain property, CSS won't do the job.
- CSS can't do computations. If you want to calculate and output a value (maybe you want to add up the numeric value of all `<price>` elements in a document), CSS won't do the job.
- CSS can't combine multiple documents. If you want to combine 53 purchase order documents and print a summary of all items ordered in those purchase orders, CSS won't do the job.



Don't take this section as a criticism of CSS; XSLT and CSS were designed for different purposes. One fairly common use of XSLT is to generate an HTML document that uses CSS. See "The XPath View of an XML Document" in Chapter 3 for an example that uses XSLT to generate CSS classes, and then uses those classes to format the HTML elements

XSLT was created to be a more powerful, flexible language for transforming documents. In this book, we go through all the features of XSLT and discuss each of them in terms of practical examples. Some of XSLT's design goals specify that:

- An XSLT stylesheet should be an XML document. This means that you can write a stylesheet that transforms a second stylesheet into another stylesheet. This kind of recursive thinking is common in XSLT.
- The XSLT language should be based on pattern matching. Most of our stylesheets consist of rules (called *templates* in XSLT) used to transform a document. Each rule says, "When you see part of a document that looks like this, here's how you convert it into something else." This is probably different from any programming you've previously done.
- XSLT should be designed to be free of side effects. In other words, XSLT is designed to be optimized so that many different stylesheet rules could be applied simultaneously. The biggest impact of this is that variables can't be modified. Once a variable is bound, you can't change its value; if variables could be changed, then processing one stylesheet rule might have side effects that impact other stylesheet rules. This is almost certainly different from any programming you've previously done.

  XSLT is heavily influenced by the design of *functional programming languages*, such as Lisp, Scheme, and Haskell. These languages also feature immutable variables. Instead of defining the templates of XSLT, functional programming languages define programs as a series of functions, each of which generates a well-defined output (free from side effects, of course) in response to a well-defined input. The goal is to execute the instructions of a given XSLT template without affecting the execution of any other XSLT template.

- Instead of looping, XSLT uses iteration and recursion. Given that variables can't be changed, how do you do something like a `for` or `do-while` loop? XSLT uses two equivalent techniques: iteration and recursion. *Iteration* means that you can write an XSLT template that says, "Get all the things that look like this, and here's what I want you to do with each of them." Although that's different from a `do-while` loop, usually what you do in a procedural language is something like, "Do this while there are any items left to process." In that case, iteration does exactly what you want.

  Recursion takes some getting used to. If you must implement something like a `for` statement (`for i=1 to 10 do`, for example), recursion is the way to go. There are a number of examples of recursion throughout the book; you can flip ahead to "Using Recursion to Do Most Anything" in Chapter 5 for more information.

Given these design goals, what are XSLT's strengths? Here are some scenarios:

- Your web site needs to deliver information to a variety of devices. You need to support ordinary desktop browsers, as well as pagers, mobile phones, and other low-resolution, low-function devices. It would be great if you could create your information in structured documents, then transform those documents into all the formats you need.
- You need to exchange data with your partners, but all of you use different database systems. It would be great if you could define a common XML data format, then transform documents written in that format into the import files you need (SQL statements, comma-separated values, etc.).
- To stay on the cutting edge, your web site gets a complete visual redesign every few months. Even though things such as server-side includes and CSS can help, they can't do everything. It would be great if your data were in a flexible format that could be transformed into any look and feel, simplifying the redesign process.
- You have documents in several different formats. All the documents are machine-readable, but it's a hassle to write programs to parse and process all of them. It would be great if you could combine all of the documents into a single format, then generate summary documents and reports based on that collection of documents. It would be even better if the report could contain calculated values, automatically generated graphics, and formatting for high-quality printing.

Throughout the book, we'll demonstrate XSLT solutions for problems just like these. Most chapters focus on particular techniques, such as sorting, grouping, and generating links between pieces of data, although we'll start with a gentle introduction to the basics.

[2.0] The Design of XSLT 2.0

XSLT 2.0 is a major enhancement to the language. XSLT 2.0 uses XPath 2.0, which itself went through many significant changes. The gap between XSLT 1.0/XPath 1.0

and XSLT 2.0/XPath 2.0 was a little over seven years (November 16, 1999 to January 23, 2007). There were two major requirements that led to the monumental amount of work required to create XSLT 2.0 and XPath 2.0:

Support for XML Schema
: XSLT and XPath now support XML Schema, which means nodes and variables can have datatypes. We can define a value to be of type `xs:dateTime`, and the XSLT processor will enforce that requirement. All XSLT 2.0 processors support the basic XML Schema datatypes. A *schema-aware processor* also supports custom datatypes. If we have a datatype named `purchaseOrder`, we can use a schema-aware processor to work with values of that type.

Integration with XQuery
: The initial work for XQuery began in 1998, and version 1.0 became a W3C Recommendation on January 23, 2007. XQuery 1.0 and XPath 2.0 share a common data model, functions, and operators. Coordinating the efforts of the XQuery, XPath, and XSLT working groups must have been a challenge.

The birthing pains of XSLT 2.0 and XPath 2.0 are behind us now, and we have a more powerful language for transforming documents. We'll discuss the changes to the language as they're relevant to our discussion of common tasks that you'll probably want to do with XSLT. All of the technical details are covered in the appendixes.

XML Basics

Almost everything we do in this book deals with XML documents. XSLT stylesheets are XML documents themselves, and they're designed to transform an XML document into something else. If you don't have much experience with XML, we'll review the basics here. For more information on XML, check out Erik T. Ray's *Learning XML* (O'Reilly, 2001) and Elliotte Rusty Harold and W. Scott Means's *XML in a Nutshell* (O'Reilly, 2001).

XML's Heritage

XML's heritage is in the Standard Generalized Markup Language (SGML). Created by Dr. Charles Goldfarb in the 1970s, SGML is widely used in high-end publishing systems. Unfortunately, SGML's perceived complexity prevented its widespread adoption across the industry (SGML also stands for "sounds great, maybe later"). SGML got a boost when Tim Berners-Lee based HTML on SGML. Overnight, the whole computing industry was using a markup language to build documents and applications.

The problem with HTML is that its tags were designed for the interaction between humans and machines. When the Web was invented in the late 1980s, that was just fine. As the Web moved into all aspects of our lives, HTML was asked to do lots of strange things. We've all built HTML pages with awkward table structures, 1-pixel

GIFs, and other nonsense just to get the page to look right in the browser. XML is designed to get us out of this rut and back into the world of structured documents.

Whatever its limitations, HTML is the most popular markup language ever created. Given its popularity, why do we need XML? Consider this extremely informative HTML element:

```
<td>12304</td>
```

What does this fascinating piece of content represent?

- Is it the postal code for Schenectady, New York?
- Is it the number of light bulbs replaced each month in Las Vegas?
- Is it the number of Volkswagens sold in Hong Kong last year?
- Is it the number of tons of steel in the Sydney Harbour Bridge?

The answer: maybe, maybe not. The point of this silly example is that there's no structure to this data. Even if we include the entire table, it takes intelligence (real, live intelligence, the kind between your ears) to make sense of this. If you saw this cell in a table next to another cell that contained the text "Schenectady," and the heading above the table read "Postal Codes for the State of New York," then as a human being, you could interpret the contents of this cell correctly. On the other hand, if you wanted to write a piece of code that took any HTML table and attempted to determine whether any of the cells in the table contained postal codes, you'd find that difficult, to say the least.

Most HTML pages have one goal in mind: the appearance of the document. Veterans of the markup industry know that this is definitely not the way to create content. The *separation of content and presentation* is a long-established tenet of the publishing industry; unfortunately, most HTML pages aren't even close to approaching this ideal. An XML document should contain information, marked up with tags that describe what all the pieces of information are, as well as the relationship between those items. Presenting the document (also known as *rendering*) involves rules and decisions separate from the document itself. As we work through dozens of sample documents and applications, you'll see how delaying the rendering decisions as long as possible has significant advantages.

Let's look at another marked-up document. Consider this:

```
<?xml version="1.0"?>
<postalcodes>
  <title>Most-used postal codes in November 2000</title>
  <item>
    <city>Schenectady</city>
    <postalcode>12304</postalcode>
    <usage-count>2039</usage-count>
  </item>
  <item>
    <city>Kuala Lumpur</city>
    <postalcode>57000</postalcode>
```

```
    <usage-count>1983</usage-count>
  </item>
  <item>
    <city>London</city>
    <postalcode>SW1P 4RG</postalcode>
    <usage-count>1722</usage-count>
  </item>
  ...
</postalcodes>
```

Although we're still in the realm of contrived examples, it would be fairly easy to write a piece of code to find the postal codes in any document that used this set of tags (as opposed to HTML's `<table>`, `<tr>`, `<td>`, etc.). Our code would look for the contents of any `<postalcode>` elements in the document. (Not to get ahead of ourselves here, but writing an XSLT stylesheet to do this might take all of 30 minutes, including a 25-minute nap.) A well-designed XML document identifies each piece of data in the document and models the relationships between those pieces of data. This means we can be confident that we're processing an XML document correctly.

Again, the key idea here is that we're separating content from presentation. Our XML document clearly delineates the pieces of data and puts them into a format we can parse easily. In this book, we illustrate a number of techniques for transforming this XML document into a variety of formats. Among other things, we can transform the item `<postalcode>12304</postalcode>` into `<td>12304</td>`.

XML Document Rules

Continuing our trip through the basics of XML, there are several rules you need to keep in mind when creating XML documents. All stylesheets we develop in this book are themselves XML documents, so all the rules of XML documents apply to everything we do. The rules are pretty simple, even though the vast majority of HTML documents don't follow them.

One important point: the XML 1.0 specification makes it clear that when an XML parser finds an XML document that breaks the rules, the parser is supposed to throw an exception and stop. The parser is not allowed to guess what the document structure should actually be. This specification avoids recreating the HTML world, where lots of ugly documents are still rendered by the average browser.

An XML document must be contained in a single element

The first element in your XML document must contain the entire document. That first element is called the *document element* or the *root element*. If more than one document element is in the document, the XML parser throws an exception. This XML document is perfectly legal:

```
<?xml version="1.0"?>
<greeting>
```

```
  Hello, World!
</greeting>
```

To be precise, this document is well-formed. XML documents are described as *well-formed* and *valid* (we'll define those terms in a minute). This XML document isn't legal at all:

```
<?xml version="1.0"?>
<greeting>
  Hello, World!
</greeting>
<greeting>
  Hey, Y'all!
</greeting>
```

There are two root elements in this document, so an XML parser refuses to process it. Also, be aware that the XML declaration (the `<?xml version="1.0"?>` part; more on this later) isn't an element at all.

All elements must be nested

If you start one element inside another, you have to end it there, too. An HTML browser is happy to render this document:

```
<b>I really, <i>really</b> like XML.</i>
```

But an XML parser will throw an exception when it sees this document. If you want the same effect, you would need to code this:

```
<b>I really, <i>really</i></b><i> like XML.</i>
```

All attributes must be quoted

You can quote the attributes with either single or double quotes. These two XML tags are equivalent:

```
<a href="http://www.oreilly.com">
<a href='http://www.oreilly.com'>
```

If you need to define an attribute that contains single or double quotes, you can use one style of quote inside the other. If you need both single and double quotes in an attribute, use the predefined entities `"` for double quotes and use `'` for single quotes:

```
<book title="XSLT, Second Edition" publisher="O'Reilly/">
<book title="XSLT, Second Edition" publisher='O&apos;reilly"/>
```

One more note: XML doesn't allow attributes without values. In other words, HTML elements such as `<ol compact>` aren't valid in XML. To code this element in XML, you'd have to give the attribute a value, as in `<ol compact="compact">`. (You have to do things this way in XHTML as well.)

XML tags are case-sensitive

In HTML, <h1> and <H1> are the same. In XML, they're not. If you try to end an <h1> element with </H1>, the parser will throw an exception.

All end tags are required

This is another area where most HTML documents break. Your browser doesn't care whether you don't have a </p> or </br> tag, but your XML parser does.

Empty tags can contain the end marker

In other words, these two XML fragments are identical:

```
<lily age="13"></lily>

<lily age="13"/>
```

Notice that there is nothing, not even whitespace, between the start tag and the end tag in the first example; that's what makes this an empty tag.

XML declarations

Some XML documents begin with an *XML declaration*, which is a line similar to this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

If no encoding is specified, the XML parser assumes you're using UTF-8 or UTF-16. UTF, the Unicode Transformation Format, is a Unicode standard that uses different numbers of bytes to represent virtually every character and ideograph from the world's languages. Be aware that each parser supports a different set of encodings, so you need to check your parser's documentation to find out what your options are.

Document Type Definitions (DTDs) and XML Schemas

All of the rules we've discussed so far apply to all XML documents. In addition, you can use DTDs and Schemas to define other constraints for your XML documents. DTDs and Schemas are metalanguages that let you define the characteristics of an XML vocabulary. For example, you might want to specify that any XML document describing a purchase order must begin with a <po> element, and the <po> element in turn contains a <customer-id> element, one or more <item-ordered> elements, and an <order-date> element. In addition, each <item-ordered> element must contain a part-number attribute and a quantity attribute.

Here's a sample DTD that defines the constraints we just mentioned:

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT po (customer-id , item-ordered+ , order-date)>

<!ELEMENT customer-id (#PCDATA)>
```

```
<!ELEMENT item-ordered EMPTY>

<!ATTLIST item-ordered  part-number CDATA  #REQUIRED
                        quantity    CDATA  #REQUIRED >
<!ELEMENT order-date EMPTY>

<!ATTLIST order-date  day   CDATA  #REQUIRED
                      month CDATA  #REQUIRED
                      year  CDATA  #REQUIRED >
```

And here's an XML Schema that defines the same document type:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="po">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="customer-id"/>
        <xsd:element ref="item-ordered" maxOccurs="unbounded"/>
        <xsd:element ref="order-date"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="customer-id" type="xsd:string"/>

  <xsd:element name="item-ordered">
    <xsd:complexType>
      <xsd:attribute name="part-number" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[0-9]{5}(-[0-9]{4})?"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="quantity" use="required" type="xsd:integer"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="order-date">
    <xsd:complexType>
      <xsd:attribute name="day" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:positiveInteger">
            <xsd:maxInclusive value="31"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="month" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:positiveInteger">
            <xsd:maxInclusive value="12"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
```

```
      <xsd:attribute name="year" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:gYear">
            <xsd:maxInclusive value="2100"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
  </xsd:schema>
```

Schemas have two significant advantages over DTDs:

They can define datatypes and other complex structures that are difficult or impossible to do in a DTD

In the previous example, we defined various constraints for the data in our XML documents. We defined that the `day` attribute must be an integer between 1 and 31, and the `month` attribute must be an integer between 1 and 12. We also used a regular expression to define a `part-number` attribute as a five-digit number, optionally followed by a dash and a four-digit number. None of those things are possible in a DTD. Schemas are far more powerful than DTDs; see Appendix D for an overview of schemas and what they can do.

Schemas are themselves XML documents

Since they are XML documents, we can write XSLT stylesheets to manipulate them. For example, it would be useful to create a graphical representation of an XML Schema. We could create a hierarchical diagram to indicate which elements could appear inside other element. XML Schema also provides the `<xsd:annotation>` and `<xsd:documentation>` elements. Those elements let us add as much documentation as we want inside the schema itself. We could then use a stylesheet to transform the schema into an HTML document or PDF file, using the relationships between elements, attributes, datatypes, and other information to generate highly structured information.

The best way to define the `<order-date>` attribute would be to use the XML Schema `xsd:date` datatype:

```
<xsd:element name="order-date" type="xsd:date"/>
```

In the DTD, we separated the date into three parts so it could be sorted or formatted in different ways. With the `xsd:date` datatype, the schema ensures that the date is valid; we can use a variety of functions to sort or format the date in different ways. (We'll discuss those functions in "[2.0] Formatting Dates and Times" in Chapter 4.)

Well-formed versus valid documents

Any XML document that follows the rules described here is said to be *well-formed*. In addition, if an XML document references a set of rules that define how the document

is structured (either a DTD or an XML Schema), and it follows all those rules, it is said to be a *valid* document.

All valid documents are well-formed; on the other hand, not all well-formed documents are valid.

Be aware that XML Schema validation can be done partially; XML Schema allows us to define parts of the document that should not be validated at all. On the other hand, DTD validation fails if any part of an XML document doesn't match the DTD.

Tags versus elements

Although many people use the two terms interchangably, a tag is different from an element. A *tag* is the text between (and including) the angle brackets (< and >). There are start tags, end tags, and empty tags. A tag consists of an element name and, if it is a start tag or an empty tag, some optional attributes. (Unlike other markup languages, end tags in XML cannot contain attributes.) An *element* consists of its start and end tags and everything in between. This might include text, other elements, and comments, as well as other things such as entity references and processing instructions.

Namespaces

A final XML topic we'll mention here is *namespaces*. Namespaces are designed to distinguish between two tags that have the same name. For example, if we have an online bookstore, we could design an XML vocabulary for books. When we ship an order to a customer, the postal service requires the customer's address to be in a certain format. It's likely that both vocabularies will define a `<title>` element. Our `<title>` element refers to the title of a book, while the shipping company's `<title>` element refers to the courtesy title of a customer (Mr., Ms., Mrs., etc.). An XML order document refers to both books and customers, so we'll use a namespace to distinguish between the two `<title>` elements. Namespaces are declared as follows:

```
<xyz xmlns:books="http://www.myco.com/books"
     xmlns:addr="http://www.usps.com/addresses">
```

In this example, the `xmlns:books` attribute associates the prefix `books` with one namespace, and the `xmlns:addr` attribute associates the `paintings` prefix with another namespace. This means that a `title` element from the `books` namespace would be coded as `<books:title>`, while a `title` element from the `addr` namespace would be referred to as `<addr:title>`.

I mention namespaces here primarily because all XSLT elements we use in this book are prefixed with the `xsl` namespace prefix. All stylesheets we write begin like this:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

(Obviously a stylesheet that uses the features of XSLT 2.0 starts with `version="2.0"`.) This opening associates the `xsl` namespace prefix with the string `http://www.w3.org/1999/XSL/Transform`. The value of the namespace prefix doesn't matter; we could start our stylesheets like this:

```
<?xml version="1.0"?>
<pdq:stylesheet version="1.0"
  xmlns:pdq="http://www.w3.org/1999/XSL/Transform">
```

What matters is the string to which the namespace prefix is mapped. Also keep in mind that all XSLT stylesheets use namespace prefixes to process the XML elements they contain. By default, anything that doesn't use the `xsl` namespace prefix is not processed—instead, it's written to the result tree. We'll discuss these topics in more detail as we go through the book.

[2.0] Datatypes

XSLT 2.0 provides support for most of the datatypes defined in XML Schema. XSLT 2.0 also defines new datatypes for durations. For example, we can define an XSLT variable and specify that its datatype is `xs:integer` or `xs:dateTime`. If we're using a schema-aware XSLT 2.0 processor, we can define our own datatypes and use those just like all the datatypes defined by XML Schema and XSLT 2.0. We cover datatypes and schemas in Chapter 3.

Programming Interfaces for XML: DOM, SAX, and Others

The two most popular APIs used to parse XML documents are the *Document Object Model* (DOM) and the *Simple API for XML* (SAX). DOM is an official recommendation of the W3C (available at *http://www.w3.org/TR/REC-DOM-Level-1*), while SAX is a de facto standard created by David Megginson and others on the XML-DEV mailing list (*http://lists.xml.org/archives*). We'll discuss these two APIs briefly here. We won't use them much in this book, but learning more about them will give you some insight into how most XSLT processors work.

See *http://www.saxproject.org/* for the SAX standard. If you'd like to learn more about the XML-DEV mailing list, send email to *mailto:xml-dev-subscribe@lists.xml.org*. You can also check out *http://lists.xml.org/archives/xml-dev/* to see the XML-DEV mailing list archives.

DOM

DOM is designed to build a tree view of your document. Remember that all XML documents must be contained in a single element. That single element then becomes the root of the tree. The DOM specification defines several language-neutral interfaces, described here:

`Node`
: This interface is the base datatype of the DOM. `Document`, `Element`, `Attr`, `Text`, `Comment`, and `ProcessingInstruction` all extend the `Node` interface.

`Document`
: This object contains the DOM representation of the XML document. Given a `Document` object, you can get the root of the tree (the `Document` element); from the root, you can move through the tree to find all elements, attributes, text, comments, processing instructions, etc. in the XML document.

`Element`
: This interface represents an element in an XML document.

`Attr`
: This interface represents an attribute of an element in an XML document.

`Text`
: This interface represents a piece of text from the XML document. Any text in your XML document becomes a `Text` node. This means that the text of a DOM object is a child of the object, not a property of it. The text of an `Element` is represented as a `Text` child of an `Element` object; the text of an `Attr` is also represented that way.

`Comment`
: This interface represents a comment in the XML document. A comment begins with `<!--` and ends with `-->`. The only restriction on its contents is that two consecutive hyphens (`--`) can appear only at the start or end of the comment. Other than that, a comment can include anything, such as angle brackets (`< >`), ampersands (`&`), and single or double quotation marks (`' "`).

`ProcessingInstruction`
: This interface represents a processing instruction in the XML document. Processing instructions look like this:

```
<?xml-stylesheet href="case-study.xsl" type="text/xsl"?>
```

  Processing instructions contain processor-specific information. The PI here (*PI* is XML jargon—feel free to drop this into casual conversations to impress your friends) is the standard way to associate an XSLT stylesheet with an XML document (more on this in a minute).

When you parse an XML document with a DOM parser, it:

- Creates objects (`Elements`, `Attr`, `Text`, `Comments`) representing the contents of the document. These objects implement the interfaces defined in the DOM specification.
- Arranges these objects in a tree. Each `Element` in the XML document has some properties (such as the element's name) and may also have some children.
- Parses the entire document before control returns to your code. This means that for large documents, there is a long delay while the document is parsed.

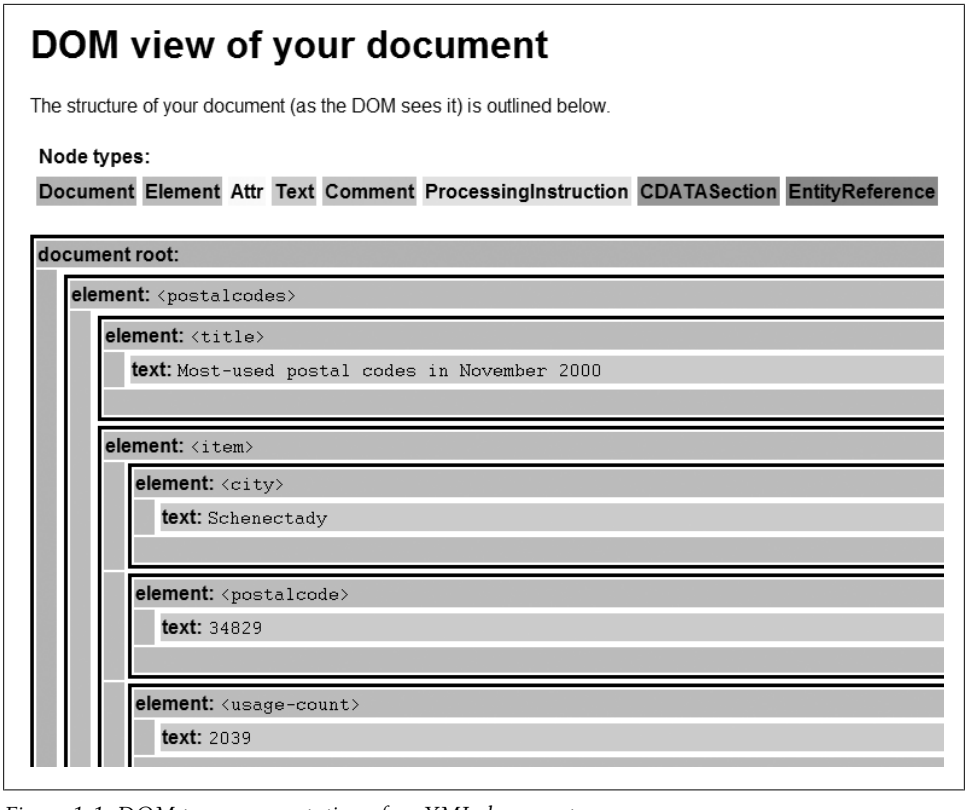
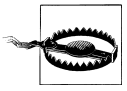


Figure 1-1. DOM tree representation of an XML document

The most significant thing about the DOM is that it is based on a tree view of your document. An XSLT processor uses a very similar tree view (with some slight differences, such as the fact that not everything we deal with in XPath and XSLT has the same root element). Understanding how a DOM parser works makes it easier to understand how an XSLT processor views your document.

A sample DOM tree. DOM, XSLT, and XPath all use tree structures to represent data from an XML document. For this reason, it's important to have at least a casual knowledge of how DOM builds a tree structure. Our earlier `<postalcodes>` document is shown as a DOM tree in Figure 1-1.

If we want to perform tasks such as find different parts of our XML document, sort the subtrees based on the first character of the text of the `<postalcode>` element, or select only the subtrees in which the text of the `<usage-count>` element has a numeric value greater than 500, we have to start at the top of the DOM tree and work our way down through the root element's descendants. When we write XSLT stylesheets, we also start at the root of the tree and work our way down.



To be honest, the DOM tree built for our document is more complicated than our beautiful picture indicates. The whitespace characters in our document (carriage return/line feed, tabs, spaces, etc.) become `Text` nodes. Normally it's a good idea to remove this whitespace so the DOM tree won't be littered with these useless nodes, but I include them here to give you a sense of the XML document's structure.

SAX

The Simple API for XML was developed by David Megginson and others on the XML-DEV mailing list. It has several important differences from DOM:

- The SAX API is interactive. As a SAX parser processes your document, it sends events to your code. You don't have to wait for the parser to finish the entire document as you do with the DOM; you get events from the parser immediately. These events let you know when the parser finds the start of the document, the start of an element, some text, the end of an element, a processing instruction, the end of the document, etc.
- SAX is designed to avoid the large memory footprint of DOM. In the SAX world, you're told when the parser finds things in the XML document; it's up to you to save those things. If you don't do anything to store the data found by the parser, it goes into the bit bucket.
- SAX doesn't provide the hierarchical view of the document that DOM does. If you need to know a lot about the structure of an XML document and the context of a given element, SAX isn't much help. Each SAX event is stateless; that is, a SAX event won't tell you, "Here's some text for the `<postalcode>` element I mentioned earlier." A SAX parser only tells you, "Here's some text." If you need to know about an XML document's structure, you have to keep track of that information yourself.

The best thing about SAX is that it is interactive. Most of the transformations currently done with XSLT take place on the server. As of this writing, most XSLT processors are based on DOM parsers. In the near future, however, we'll see XSLT processors based on SAX parsers. This means that the processor can start generating results almost as soon as the parse of the source document begins, resulting in better throughput and creating the perception of faster service. Because DOM, XPath, and XSLT all use trees to represent XML documents, DOM is more relevant to our discussions here. Nevertheless, it's useful to know how SAX parsers work, especially as SAX-based XSLT processors begin to rear their speedy little heads.

Other programming interfaces

There are a number of other XML programming interfaces, including JDOM, DOM4J, and StAX. These have two important characteristics:

In-memory versus event-driven
: In-memory interfaces, such as DOM, create data structures that represent the XML document. Event-driven interfaces, such as SAX, receive data from the parser as it parses the document.

Push versus pull
: A push interface pushes data from the parser to the application. When the parser has some data, it uses a callback interface to push that data to the application. SAX is an example of a push interface. On the other hand, a pull interface is still event-driven, but the application tells the parser when it wants the next event. StAX, the Streaming API for XML, is an example of a pull interface. (StAX is also known as JSR 173.)

There are two other approaches we'll mention briefly. In *data binding*, an XML document is transformed into an object. The contents of the original XML document are represented as the properties of that object. Finally, a new parsing technique called *non-extractive XML processing* creates Virtual Token Descriptors that contain the offset, length, and other information of XML tokens inside the XML file itself.

The Wikipedia entry *http://en.wikipedia.org/wiki/XML#Processing_XML_files* has more detail on these approaches as well as links to various tools that implement them.

XSLT Standards

XSLT 1.0 is defined in two documents: the XSLT and XPath specifications. XSLT 2.0 and XPath 2.0, on the other hand, are defined in a set of eight documents. We'll discuss all of those specifications briefly in the next section.

XSL transformations (XSLT) version 1.0

The original standard became a recommendation of the W3C on November 16, 1999. The spec lives here: *http://www.w3.org/TR/xslt*.

XML path language (XPath) version 1.0

XPath 1.0 became a standard on the same day as XSLT 1.0. XPath began as part of XSLT. If we're going to write a stylesheet to transform an XML document, we have to have a syntax for describing different parts of that document. As the development of XSLT continued, it became obvious that XPath was useful for a variety of applications, so XPath became a separate standard. You can find the definition of XPath 1.0 at *http://www.w3.org/TR/xpath*.

XSL transformations (XSLT) version 2.0

The basic definition of XSLT 2.0 is at *http://www.w3.org/TR/xslt20/*. This document defines the elements of XSLT 2.0 and a variety of functions and also defines how XSLT 2.0 processes an XML document.

XML path language (XPath) version 2.0

The basic definition of XPath 2.0 is at *http://www.w3.org/TR/xpath20/*. XPath 2.0 is built on top of several other documents; we'll list those next.

XQuery 1.0 and XPath 2.0 Data Model (XDM)

This spec defines the way XPath 2.0, XSLT 2.0, and XQuery 1.0 organize data. It defines the information contained in the input to an XSLT 2.0 or XQuery 1.0 processor. It also defines all of the legal values for expressions in XPath 2.0, XSLT 2.0, and XQuery 1.0. You can find the spec at *http://www.w3.org/TR/xpath-datamodel/*.

XQuery 1.0 and XPath 2.0 functions and operators

This spec, also known as F&O, defines all of the functions and data operators available in XPath 2.0 and XQuery 1.0. For example, the spec defines how an `xs:yearMonthDuration` can be divided by an `xs:double` value. It also defines the `matches( )` function, which determines if a value matches a regular expression. The spec is available at *http://www.w3.org/TR/xpath-functions/*.

XQuery 1.0 and XPath 2.0 formal semantics

The formal semantics spec defines a precise meaning to all of the legal expressions in XPath 2.0 and XQuery 1.0. The XQuery 1.0 and XPath 2.0 Data Model is used in those precise definitions. Possibly the least useful spec to XSLT programmers, it's available at *http://www.w3.org/TR/xquery-semantics/*.

XSLT 2.0 and XQuery 1.0 serialization

The serialization spec defines how to take an instance of the XQuery 1.0/XPath 2.0 Data Model and serialize it. For the examples in this book, we'll usually take the results generated by our XSLT stylesheet and write them to a file; the serialization spec defines how that process works. The spec is available at *http://www.w3.org/TR/xslt-xquery-serialization/*.

XQuery 1.0: an XML query language

XQuery 1.0 is a separate language that is based on XPath and other query languages. It is a superset of XPath 2.0. We won't cover XQuery in any detail in this book, but be aware that the data model, the functions, and the operators of XPath 2.0 are shared by XQuery. See *http://www.w3.org/TR/xquery/* for the complete details.

XML syntax for XQuery 1.0 (XQueryX)

One of the requirements of the XQuery working group was to provide an XML syntax for the language. XQueryX provides that syntax. It maps the XQuery grammar into XML tags. As such, it is not particularly easy or convenient for humans, but it can be

very useful for various tools and utilities. The spec is available at *http://www.w3.org/TR/xqueryx*.

XML Standards

When we talk about writing stylesheets, we'll work with two standards: XSLT and XPath. XSLT defines a set of primitives used to describe a document transformation, while XPath defines a syntax for describing locations in XML documents. When we write stylesheets, we'll use XSLT to tell the processor what to do, and we'll use XPath to tell the processor what document to do it to. Both standards are available at the W3C's web site; see *http://www.w3.org/TR/xslt* and *http://www.w3.org/TR/xpath* for more information.

There are other XML-related standards, of course. We'll discuss them here briefly, with a short mention of how (or whether) they relate to our work with XSLT and XPath.

XML 1.0

The foundation upon which everything else is built. See *http://www.w3.org/TR/REC-xml*.

XML 1.1

You can find the XML 1.1 standard at *http://www.w3.org/TR/xml11/*.

The Extensible Stylesheet Language (XSL)

Also called the *Formatting Objects specification* or *XSL-FO*, this standard deals with rendering XML elements. Although most people think of rendering as formatting for a browser or a printed page, researchers use the specification to render XML elements as Braille or as audio files. (That being said, the main market for this technology is in producing high-quality printed output.) As of this writing, the latest version of XSL is 1.1. A couple of the examples in this book use formatting objects and the Apache XML Project's Formatting Object to PDF translator (FOP) tool; see *http://xml.apache.org/fop* for more information on FOP. For more information on XSL, see *http://www.w3.org/TR/xsl*.

XML Schemas

In our earlier examples, we had a brief example of an XML Schema. Part 1 of the specification deals with XML document structures; it contains XML elements that define what can appear in an XML document. You use these elements to specify which elements can be nested inside others, how many times each element can appear, the attributes of those elements, and other features. Part 2 of the specification defines basic datatypes used in XML Schemas and rules for deriving new datatypes from existing ones.

The two specifications are available at *http://www.w3.org/TR/xmlschema-1* and *http://www.w3.org/TR/xmlschema-2*. For a good introduction to XML Schemas, see the XML Schema Primer, available at *http://www.w3.org/TR/xmlschema-0*.

RelaxNG

RelaxNG is a simple schema language designed as an alternative to XML Schema. One significant difference between the two is that RelaxNG avoids the many datatype definitions of XML Schema. With RelaxNG, you validate an XML document with datatype definitions imported from elsewhere (including XML Schema, for example). The home page of the OASIS RelaxNG committee is here: *http://www.oasis-open.org/committees/relax-ng/*. You can find the latest version of the spec as well as a tutorial there.

Schematron

Schematron is an elegant way to validate documents. It has a simple syntax (only six elements) and uses XPath to specify patterns in XML documents. The most interesting and most widely used implementation of Schematron is written in XSLT. For more information, including a link to the latest version of the ISO standard for Schematron, visit *http://www.schematron.com/*.

The Simple API for XML (SAX)

The SAX API defines the events and interfaces used to interact with a SAX parser. SAX and DOM are the most common APIs used to work with XML documents. See *http://www.saxproject.org/* for the complete specification.

Document Object Model (DOM)

The DOM, as we discussed earlier, is a programming API for documents. It defines a set of interfaces and methods used to view an XML document as a tree structure. XSLT and XPath use a similar tree view of XML documents. The home of the DOM is *http://www.w3.org/DOM/*. This page contains links to all of the W3C Recommendations (Levels 1, 2, and 3) and related documents. The DOM doesn't affect what we'll do here, but it's useful to have a passing knowledge of it. (The XPath data model is similar to the DOM.)

Namespaces in XML

As we mentioned earlier, namespaces provide a way to avoid name collisions when two XML elements have the same name. See *http://www.w3.org/TR/REC-xml-names/* for the version 1.0 spec; version 1.1 is at *http://www.w3.org/TR/REC-xml-names11/*.

Associating stylesheets with XML documents

It's possible to reference an XSLT stylesheet within an XML document. This specification uses processing instructions to define one or more stylesheets that should be

used to transform an XML document. You can define different stylesheets to be used for different browsers. See *http://www.w3.org/TR/xml-stylesheet* for complete information. Here's the start of an XML document, with two associated stylesheets:

```
<?xml version="1.0"?>
<?xml-stylesheet href="docbook/html/docbook.xsl" type="text/xsl"?>
<?xml-stylesheet href="docbook/wap/docbook.xsl"  type="text/xsl" media="wap"?>
```

In this example, the first stylesheet is the default because it doesn't have a `media` attribute. The second stylesheet will be used when the `User-Agent` field from the HTTP header contains the string `wap`, identifying the requester of a document as a WAP browser. The advantage of this technique is that you can define several different stylesheets within a particular document and have each stylesheet generate useful results for different browser or client types. The disadvantage of this technique is that we're effectively putting rendering instructions into our XML document, something we prefer to avoid.

Scalable Vector Graphics (SVG)

The SVG specification defines an XML vocabulary for vector graphics. Described by some as "PostScript with angle brackets," it allows you to define images that can be scaled to any size or resolution. See *http://www.w3.org/TR/SVG/* for details.

XML pointer language (XPointer) version 1.0

XPointer provides a way to identify a fragment of a web resource. It uses XPath to identify fragments. The XPointer Framework is defined at *http://www.w3.org/TR/xptr-framework/*.

XML linking language (XLink) version 1.0

XLink defines an XML vocabulary for linking to other web resources within an XML document. It supports the unidirectional links we're all familiar with in HTML, as well as more sophisticated links. See *http://www.w3.org/TR/xlink/*.

Installing XSLT Processors

Before we dive in to creating stylesheets, we'll cover how to install four popular XSLT processors.

Installing Xalan

In this section, we'll go over how to install the Xalan XSLT processor. In the next chapter, we'll create our first stylesheet and use it to transform an XML document.

The installation process is pretty simple, assuming you already have a Java Runtime Environment (JRE) installed on your machine. Although very little of the code we look

at in this book uses Java, the Xalan XSLT processor itself is written in Java. Once you've installed the JRE, go to *http://xml.apache.org/xalan-j/* and download the latest stable build of the code. (If you're feeling brave, feel free to download last night's build instead.)

Once the Xalan *.zip* or *.gzip* file is downloaded, unpack it and add three files to your `CLASSPATH`. The three files include two *.jar* files for the Xerces parser, and the *.jar* file for the Xalan stylesheet engine itself. As of this writing, the *.jar* files are named *xalan.jar*, *xercesImpl.jar*, and *xml-apis.jar*. (There's a fourth file, *bsf.jar*, that includes the Bean Scripting Framework, but we'll use that for extensions only.)

To make sure Xalan is installed correctly, go to a command prompt and type the following command:

```
java org.apache.xalan.xslt.Process
```

This is a Java class, so everything is case-sensitive. You should see an error message like this:

```
java org.apache.xalan.xslt.Process
=xslproc options:
    -IN inputXMLURL
   [-XSL XSLTransformationURL]
   [-OUT outputURL]
   [-LXCIN compiledStylesheetFileNameIn]
   [-LXCOUT compiledStylesheetFileNameOutOut]
...
```

If you get this message, you're all set! You're ready for the next chapter, in which we'll build our very first XSLT stylesheet.

Installing Saxon

As of this writing, the most complete open source XSLT 2.0 stylesheet processor is Saxon. Written by Michael Kay, the editor of the XSLT 2.0 spec, it is available at *http://saxon.sourceforge.net*. When you download the file (currently *saxonb9-0-0-2j.zip*), add *saxon9.jar* to your `CLASSPATH`. There are also nine other files, *saxon9-ant.jar*, *saxon9-dom.jar*, *saxon9-dom4j.jar*, *saxon9-jdom.jar*, *saxon9-s9api.jar*, *saxon9-sql.jar*, *saxon9-xom.jar*, *saxon9-xpath.jar*, and *saxon9-xqj.jar*. These *.jar* files enable additional functions; see the Saxon documentation for more information about them. For most of what we'll do in this book, *saxon9.jar* is all you'll need.

Once you've installed Saxon and updated your classpath, go to a command prompt and type the following command:

```
java net.sf.saxon.Transform
```

You should get a message like this:

```
No source file name
Saxon 9.0.0.3J from Saxonica
Usage: see http://www.saxonica.com/documentation/using-xsl/commandline.html
```

```
Options:
  -a                    Use xml-stylesheet PI, not style-doc argument
  -c:filename           Use compiled stylesheet from file
  -cr:classname         Use collection URI resolver class
  -dtd:on|off           Validate using DTD
  -expand:on|off        Expand defaults defined in schema/DTD
  -explain[:filename]   Display compiled expression tree
  -ext:on|off           Allow|Disallow external Java functions
  -im:modename          Initial mode
  -it:template          Initial template
  -l:on|off             Line numbering for source document
  -m:classname          Use message receiver class
  -o:filename           Output file or directory
  -or:classname         Use OutputURIResolver class
  -outval:recover|fatal Handling of validation errors on result document
  -p:on|off             Recognize URI query parameters
  -r:classname          Use URIResolver class
  -repeat:N             Repeat N times for performance measurement
  -s:filename           Initial source document
  -sa                   Schema-aware transformation
...
```

The error message will list dozens of options. Most of the time we'll simply specify the source XML file and the XSLT stylesheet.

Saxon is also available in a closed source version, Saxon-SA, that provides complete support for the XML Schema functions defined in XSLT 2.0, XPath 2.0, and XQuery 1.0. All of the examples in this book that use schema-aware functions were tested with the closed source, commercial version of Saxon.

To install the schema-aware version of Saxon, you need to add *saxon9sa.jar* to your `CLASSPATH`. When you purchase Saxon-SA, you'll get a *saxon-license.lic* file; put that file into the *Saxon/bin* directory and add that directory to your system `PATH`. The command to run the schema-aware version of Saxon is slightly different:

```
java com.saxonica.Transform
```

The version number will be different, but everything else should be the same:

```
No source file name
Saxon-SA 9.0.0.3J from Saxonica
Usage: see http://www.saxonica.com/documentation/using-xsl/commandline.html
Options:
  -a                    Use xml-stylesheet PI, not style-doc argument
...
```

Installing the Microsoft XSLT Processor

The most commonly used XSLT processor in the .NET world is the Microsoft XSLT processor. The best way to find the tools is to visit *http://msdn.microsoft.com/xml*. As of this writing (2008), the file you want to download is *msxsl.exe*. Put this on your system path, then go to a command prompt and type the following command:

```
msxsl
```

You'll see a message like this:

```
Microsoft (R) XSLT Processor Version 4.0

Usage: MSXSL source stylesheet [options] [param=value...] [xmlns:prefix=uri...]

Options:
    -?            Show this message
    -o filename   Write output to named file
    -m startMode  Start the transform in this mode
    -xw           Strip non-significant whitespace from source and stylesheet
...
```

In Chapter 9, we'll look at C# code that uses the XSLT processor built into the .NET framework. If we're just transforming XML documents from the command line, *msxsl.exe* is all we'll need.

Installing the Altova XSLT Engine

As of this writing (early 2008), the only zero-cost XSLT 2.0 processor that provides schema support is the Altova XSLT engine. This is the XSLT processor at the heart of Altova's XMLSpy product. It is currently a Windows-only download available under a royalty-free license at *http://www.altova.com/altovaxml*.

To install the engine, download the software (currently a setup file named *altovaxml2008.exe*) and run it. At a command prompt, type the following command:

```
altovaxml
```

You'll see a message like this:

```
AltovaXML Version 2008 sp1
Copyright (c) 1998-2007 Altova GmbH. All rights reserved.
Use of this software is subject to the license agreement at
http://www.altova.com/altovaxmldla.html


Use the xslt1 engine:
   /xslt1 <filename> /in <filename> [/param name=value] [/out <filename>]
Use the xslt2 engine:
   /xslt2 <filename> /in <filename> [/param name=value] [/out <filename>]
Use the xquery engine:
   /xquery <filename> [/in <filename>] [/param name=value] [/out <filename>]
                      [serialization options]
Use the validator:
   /validate <filename> [/schema <filename> | /dtd <filename>]
   /wellformed <filename>

Parameters:
   /validate, /v <filename>   Schema validates the specified XML file
   /wellformed, /w <filename> Check if specified XML file is well-formed
   /xslt1 <filename>          Sets the source for XSLT1 stylesheet
   /xslt2 <filename>          Sets the source for XSLT2 stylesheet
   /xquery, /xq <filename>    Sets the source for XQuery expression
```

```
/in <filename>              Sets the source for XML data
/out <filename>             Sets the output file name
                            If omitted, data is written to stdout
/param <name>=<value>       Adds external paramter
```

Summary

In this chapter, we've gone over the basics of XML and talked about DOM and SAX, two standards that are commonly used by XSLT processors. We also talked about other technology standards and how to install several stylesheet processors. At this point, you've got everything you need to build and use your first stylesheets, which we'll do in the next chapter.

CHAPTER 2

The Obligatory Hello World Example

In future chapters, we'll spend a lot of time talking about XSLT, XPath, and various advanced functions used to transform XML documents. First, though, we'll go through a short example to illustrate how stylesheets work.

Goals of This Chapter

By the end of this chapter, you should know:

- How to create a basic stylesheet
- How to use a stylesheet to transform an XML document
- How a stylesheet processor uses a stylesheet to transform an XML document
- The structure of an XSLT stylesheet

Transforming Hello World

Continuing the tradition of Hello World examples begun by Brian Kernighan and Dennis Ritchie in *The C Programming Language* (Prentice Hall, 1988), we'll transform a Hello World XML document.

Our Sample Document

First, we'll look at our sample document. This simple XML document, courtesy of the XML 1.0 specification, contains the famous friendly greeting to the world:

```
<?xml version="1.0"?>
<!-- greeting.xml -->
<greeting>
  Hello, World!
</greeting>
```

What we'd like to do is transform this fascinating document into something we can view in an ordinary household browser.

A Sample Stylesheet

Here's an XSLT stylesheet that defines how to transform the XML document:

```
<?xml version="1.0"?>
<!-- greeting.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <xsl:apply-templates select="greeting"/>
  </xsl:template>

  <xsl:template match="greeting">
    <html>
      <body>
        <h1>
          <xsl:value-of select="."/>
        </h1>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

We'll talk about these elements and what they do in just a minute. Keep in mind that the stylesheet is itself an XML document, so we have to follow all of the document rules we discussed in the previous chapter.

Transforming the XML Document

To transform the XML document using the XSLT stylesheet, run this command if you're using Xalan:

```
java org.apache.xalan.xslt.Process -in greeting.xml -xsl greeting.xsl
  -out greeting.html
```

For Saxon, the command looks like this:

```
java net.sf.saxon.Transform -o greeting.html greeting.xml greeting.xsl
```

If you're using the Schema-aware version of Saxon, the name of the Java class is different:

```
java com.saxon.Transform -o greeting.html greeting.xml greeting.xsl
```

The command for the Altova XSLT engine is:

```
altovaxml /xslt1 greeting.xsl /in greeting.xml /out greeting.html
```

Finally, if you're using Microsoft's MSXSL, type this command:

```
msxsl greeting.xml greeting.xsl -o greeting.html
```

Figure 2-1. HTML version of our Hello World file

This command transforms the document *greeting.xml*, using the templates found in the stylesheet *greeting.xsl*. The results of the transformation are written to the file *greeting.html*. Check the output file in your favorite browser to make sure the transformation worked correctly.

This is one of the few times in this book we'll cover the syntax of the command to run a transformation. The exception to this rule will be when you need to do something more advanced (pass parameters to a stylesheet, for example). Typically, all you need to know are the filenames of the XML, XSL, and output files, and the format of the command for your stylesheet processor.

Stylesheet Results

The XSLT processor generates these results:

```
<html>
<body>
<h1>
  Hello, World!
</h1>
</body>
</html>
```

When rendered in a browser, our output document looks like Figure 2-1.

Congratulations! You've now used XSLT to transform an XML document.

How a Stylesheet Is Processed

Now that we're giddy with the excitement of having transformed an XML document, let's discuss the stylesheet and how it works. A big part of the XSLT learning curve is figuring out how stylesheets are processed. To make this clear, we'll go through the steps taken by the stylesheet processor to create the HTML document we want.

Parsing the Stylesheet

Before the XSLT processor can process your stylesheet, it has to read it. Conceptually, it doesn't matter how the XSLT processor stores the information from your stylesheet.

For our purposes, we'll just assume that the XSLT processor can magically find anything it needs in our stylesheet. (If you really must know, Xalan uses an optimized table structure to represent the stylesheet; other processors may use that approach or something else.)

Our stylesheet contains three items: an `<xsl:output>` element that specifies HTML as the output format, and two `<xsl:template>` elements that specify how parts of our XML document should be transformed.

Parsing the Transformee

Now that the XSLT processor has processed the stylesheet, it needs to read the document it's supposed to transform. The XSLT processor builds a tree view from the XML source. This tree view is what we'll keep in mind when we build our stylesheets.

Lather, Rinse, Repeat

Finally, we're ready to begin the actual work of transforming the XML document. The XSLT processor may set some properties based on your stylesheet (in the previous example, it would set its output method to HTML), then it begins processing as follows:

1. Do I have any nodes to process? The nodes to process are represented by the *context*. Initially, the context is the root of the XML document, but it changes throughout the stylesheet. We'll talk about the context extensively in the next chapter. (Note: all XSLT processors enjoy being anthropomorphized, so I'll often refer to them this way.)

While any nodes are in the context, do the following:

2. Get the next node from the context. Do I have any `<xsl:template>`s that match it? (In our example, the next node is the root node, represented in XPath syntax by `/`.) There is a template that matches this node—it's the one that begins `<xsl:template match="/">`.
3. If one or more `<xsl:template>`s match, pick the right one and process it. The right one is the most specific template. For example, `<xsl:template match="/html/body/h1/p">` is more specific than `<xsl:template match="p">`. (See the discussion of `<xsl:template>` in Appendix A for more information.) If no `<xsl:template>`s match, the XSLT processor uses some built-in rules. See the section "Built-in Template Rules" later in this chapter for more information.

Notice that this is a recursive processing model. We process the current node by finding the right `xsl:template` for it. That `xsl:template` may in turn invoke other `xsl:template`s, which invoke `xsl:template`s as well. This model takes some getting used to, but it is actually quite elegant once you're accustomed to it.

If it helps, you can think of the root template (`<xsl:template match="/">`) as the `main` method in a C, C++, or Java program. No matter how much code you've written, everything starts in `main`. Similarly, no matter how many `<xsl:template>`s you've defined in your stylesheet, everything starts in `<xsl:template match="/">`.

Walking Through Our Example

Let's revisit our example and see how the XSLT processor transforms our document:

1. The XSLT stylesheet is parsed and converted into a tree structure.
2. The XML document is also parsed and converted into a tree structure. Don't worry too much about what that tree looks like or how it works; for now, just assume that the XSLT processor knows everything that's in the XML document and the XSLT stylesheet. After the first two steps are done, when we describe various things using XSLT and XPath, the processor knows what we're talking about.
3. The XSLT processor is now at the root of the XML document. This is the original context.
4. There is an `xsl:template` that matches the document root:

   ```
   <xsl:template match="/">
     <xsl:apply-templates select="greeting"/>
   </xsl:template>
   ```

   A single forward slash (/) is an *XSLT pattern* (written in *XPath*) that matches "document nodes."
5. Now the process begins again inside the `xsl:template`. Our only instruction here is to apply whatever `xsl:template`s might apply to any `greeting` elements in the current context. The current context inside this template is defined by the `match` attribute of the `xsl:template` element. This means the XSLT processor is looking for any `greeting` elements at the document root.

   Because one `greeting` element is at the document root, the XSLT processor must deal with it. (If more than one element matches in the current context, the XSLT processor deals with each one in the order in which they appear in the document; this is known as *document order*.) Looking at the `greeting` element, the `xsl:template` that applies to it is the second `xsl:template` in our stylesheet:

   ```
   <xsl:template match="greeting">
     <html>
       <body>
         <h1>
           <xsl:value-of select="."/>
         </h1>
       </body>
     </html>
   </xsl:template>
   ```

6. Now we're in the `xsl:template` for the `greeting` element. The first three elements in this `xsl:template` (`<html>`, `<body>`, and `<h1>`) are HTML elements. Because they're not defined with a namespace declaration, the XSLT processor passes those HTML elements through to the output stream unaltered.

   The middle of our `xsl:template` is an `xsl:value-of` element. This element writes the value of something to the output stream. In this case, we're using the XPath expression `.` (a single period) to indicate the current node. The XSLT processor looks at the current node (the `greeting` element we're currently processing) and outputs its text.

   Because our stylesheet is an XML document (we're really harping on that, aren't we?), we have to end the `<h1>`, `<body>`, and `<html>` elements here. At this point, we're done with this template, so control returns to the template that invoked us.
7. Now we're back in the template for the root element. We've processed all the `<greeting>` elements, so we're finished with this template.
8. No more elements are in the current context (there is only one root element), so the XSLT processor is done.

Stylesheet Structure

As the final part of our introduction to XSLT, we'll look at the contents of the stylesheet itself. We'll explain all the things in our stylesheet and discuss other approaches we could have taken.

The <xsl:stylesheet> Element

The `<xsl:stylesheet>` element is typically the root element of an XSLT stylesheet:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

First of all, the `<xsl:stylesheet>` element defines the version of XSLT we're using, along with a definition of the `xsl` namespace. To be compliant with the XSLT specification, your stylesheet should always begin with this element, coded exactly as shown here. Some stylesheet processors, notably Xalan, issue a warning message if your `<xsl:stylesheet>` element doesn't have these two attributes with these two values. For all examples in this book, we'll start the stylesheet with this exact element, defining other namespaces as needed.

The <xsl:output> Element

Next, we specify the output method. The XSLT specification defines three output methods: `xml`, `html`, and `text`. We're creating an HTML document, so HTML is the output method we want to use. In addition to these three methods, an XSLT processor

is free to define its own output methods, so check your XSLT processor's documentation to see if you have any other options:*

```
<xsl:output method="html"/>
```

A variety of attributes are used with the different output methods. For example, if you're using `method="xml"`, you can use `doctype-public` and `doctype-system` to define the public and system identifiers to be used in the the document type declaration. If you're using `method="xml"` or `method="html"`, you can use the `indent` attribute to control whether or not the output document is indented. The discussion of the `<xsl:output>` element in Appendix A has all the details.

Our First <xsl:template>

Our first template matches `"/"`, the XPath expression for the document's root element:

```
<xsl:template match="/">
  <xsl:apply-templates select="greeting"/>
</xsl:template>
```

The <xsl:template> for <greeting> Elements

The second `<xsl:template>` element processes any `<greeting>` elements in our XML source document:

```
<xsl:template match="greeting">
  <html>
    <body>
      <h1>
        <xsl:value-of select="."/>
      </h1>
    </body>
  </html>
</xsl:template>
```

Built-in Template Rules

Although most stylesheets we'll develop in this book explicitly define how various XML elements should be transformed, XSLT defines several built-in template rules that apply in the absence of any specific rules. These rules have a lower priority than any other templates, so they're always overridden when you define your own templates. The built-in templates are listed here.

* *[2.0]* XSLT 2.0 also defines the `xhtml` output method.

Built-in template rule for element and document nodes

This template processes the document node and any of its children. This processing ensures that recursive processing will continue, even if no template is declared for a given element:

```
<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>
```

As an example, given this document:

```
<?xml version="1.0"?>
<x>
  <y>
    <z/>
  </y>
</z>
```

The built-in template rule for element and document nodes means that we could write a stylesheet containing only a template with `match="z"` and the `<z>` element will still be processed, even if there are no template rules for the `<x>` and `<y>` elements.

Built-in template rule for modes

This template ensures that element and document nodes are processed, regardless of any mode that might be in effect. (See "Templates à la mode" in Chapter 5 for more information on the `mode` attribute.)

```
<xsl:template match="*|/" mode="x">
  <xsl:apply-templates mode="x"/>
</xsl:template>
```

Built-in template rule for text and attribute nodes

This template copies the text of all text and attribute nodes to the output tree. Be aware that you have to actually select the text and attribute nodes for this rule to be invoked:

```
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
```

Built-in template rule for comment and processing instruction nodes

This template does nothing:

```
<xsl:template match="comment()|processing-instruction()"/>
```

Built-in template rule for namespace nodes

This template also does nothing:

```
<xsl:template match="namespace()"/>
```

Top-Level Elements

To this point, we haven't actually talked about our source document or how we're going to transform it. We're simply setting up some properties for the transform. There are other elements we can put at the start of our stylesheet. Any element whose parent is the `<xsl:stylesheet>` element is called a *top-level element*. Here is a brief discussion of the other top-level elements:

`<xsl:include>` *and* `<xsl:import>`
: These elements refer to another stylesheet. The other stylesheet and all of its contents are included in the current stylesheet. The main difference between `<xsl:import>` and `<xsl:include>` is that a template, variable, or anything else imported with `<xsl:import>` has a lower priority than the things in the current stylesheet. This gives you a mechanism to subclass stylesheets, if you want to think about this from an object-oriented point of view. You can import another stylesheet that contains common templates, but any templates in the *importing* stylesheet will be used instead of any templates in the imported stylesheet. Another difference is that `<xsl:import>` can only appear at the beginning of a stylesheet, while `<xsl:include>` can appear anywhere.

`<xsl:strip-space>` *and* `<xsl:preserve-space>`
: These elements contain a space-separated list of elements from which whitespace should be removed or preserved in the output. To define these elements globally, use `<xsl:strip-space elements="*"/>` or `<xsl:preserve-space elements="*"/>`. If we want to specify that whitespace be removed for all elements except for `<greeting>` and `<salutation>` elements, we would add this markup to our stylesheet:

```
<xsl:strip-space elements="*"/>
<xsl:preserve-space elements="greeting
            salutation"/>
```

`<xsl:key>`
: This element defines a key, which is similar to defining an index on a database. We'll talk more about the `<xsl:key>` element and the `key( )` function in "Branching Elements of XSLT" in Chapter 5.

`<xsl:variable>`
: This element defines a variable. Any `<xsl:variable>` that appears as a top-level element is global to the entire stylesheet. Variables are discussed extensively in "Variables" in Chapter 5.

`<xsl:param>`
: This element defines a parameter. As with `<xsl:variable>`, any `<xsl:param>` that is a top-level element is global to the entire stylesheet. Parameters are discussed extensively in "Parameters" in Chapter 5.

Other stuff
: More obscure elements that can appear as top-level elements are `<xsl:decimal-format>`, `<xsl:namespace-alias>`, and `<xsl:attribute-set>`. All are discussed in Appendix A.

Other Approaches

One mantra of the Perl community is, "There's more than one way to do it." That's true with XSLT stylesheets as well. We could have written our stylesheet like this:

```
<?xml version="1.0"?>
<!-- greeting2.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <body>
        <xsl:apply-templates select="greeting"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="greeting">
    <h1>
      <xsl:value-of select="."/>
    </h1>
  </xsl:template>
</xsl:stylesheet>
```

In this version, we put the wrapper elements for the HTML document in the template for the root element. One of the things you should think about as you build your stylesheets is where to put elements such as `<html>` and `<body>`. Let's say our XML document looked like this instead:

```
<?xml version="1.0"?>
<greetings>
  <greeting>Hello, World!</greeting>
  <greeting>Hey, Y'all!</greeting>
</greetings>
```

In this case, we would have to put the `<html>` and `<body>` elements in the `<xsl:template>` for the root element. If they were in the `<xsl:template>` for the `<greeting>` element, the output document would have multiple `<html>` elements, which isn't valid in an HTML document. Our updated stylesheet would look like this:

```
<?xml version="1.0"?>
<!-- multiple-greetings.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <body>
        <xsl:apply-templates select="greetings/greeting"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="greeting">
    <h1>
      <xsl:value-of select="."/>
    </h1>
  </xsl:template>
</xsl:stylesheet>
```

Notice that we had to modify our XPath expression; what was originally `greeting` is now `greetings/greeting`. As we develop stylesheets, we'll have to make sure our XPath expressions match the document structure. When you get unexpected results, or no results, an incorrect XPath expression is usually the cause.

As a final example, we could also write our stylesheet with only one `xsl:template`:

```
<?xml version="1.0"?>
<!-- greeting3.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <body>
        <h1>
          <xsl:value-of select="greeting"/>
        </h1>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Although this is the shortest of our sample stylesheets, our examples will tend to feature a number of short templates, each of which defines a simple transform for a few elements. This approach makes your stylesheets much easier to understand, maintain, and reuse. The more transformations you cram into each `xsl:template`, the more difficult it is to debug your stylesheets, and the more difficult it is to reuse the templates elsewhere.

Sample Gallery

Before we get into more advanced topics, we'll transform our Hello World document in other ways. We'll look through simple stylesheets that convert our small XML document into the following things:

- A Scalable Vector Graphics (SVG) file
- A PDF file
- A Java program
- A Virtual Reality Modeling Language (VRML) file

The Hello World SVG File

Our first example will convert our Hello World document into an SVG file:

```
<?xml version="1.0"?>
<!-- svg-greeting.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:svg="http://www.w3.org/2000/svg">

  <xsl:template match="/">
    <svg:svg width="10cm" height="4cm">
      <svg:g>
        <svg:defs>
          <svg:radialGradient id="MyGradient"
            cx="4cm" cy="2cm" r="3cm" fx="4cm" fy="2cm">
            <svg:stop offset="0%" style="stop-color:red"/>
            <svg:stop offset="50%" style="stop-color:blue"/>
            <svg:stop offset="100%" style="stop-color:red"/>
          </svg:radialGradient>
        </svg:defs>
        <svg:rect style="fill:url(#MyGradient); stroke:black"
          x="1cm" y="1cm" width="8cm" height="2cm"/>
        <svg:text x="5.05cm" y="2.25cm" text-anchor="middle"
          style="font-family:Verdana; font-size:24;
          font-weight:bold; fill:black">
          <xsl:apply-templates select="greeting"/>
        </svg:text>
        <svg:text x="5cm" y="2.2cm" text-anchor="middle"
          style="font-family:Verdana; font-size:24;
          font-weight:bold; fill:white">
          <xsl:apply-templates select="greeting"/>
        </svg:text>
      </svg:g>
    </svg:svg>
  </xsl:template>
```

Figure 2-2. SVG version of our Hello World file

```
    <xsl:template match="greeting">
      <xsl:value-of select="."/>
    </xsl:template>

</xsl:stylesheet>
```

As you can see from this stylesheet, most of the code here simply sets up the structure of the SVG document. This is typical of many stylesheets; once you learn what the output format should be, you merely extract content from the XML source document and insert it into the output document at the correct spot. When we transform the Hello World document with this stylesheet, here are the results:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg:svg xmlns:svg="http://www.w3.org/2000/svg" width="10cm" height="4cm">
  <svg:g>
    <svg:defs>
      <svg:radialGradient id="MyGradient" cx="4cm" cy="2cm" r="3cm"
           fx="4cm" fy="2cm">
        <svg:stop offset="0%" style="stop-color:red"/>
        <svg:stop offset="50%" style="stop-color:blue"/>
        <svg:stop offset="100%" style="stop-color:red"/>
      </svg:radialGradient>
    </svg:defs>
    <svg:rect style="fill:url(#MyGradient); stroke:black" x="1cm"
      y="1cm" width="8cm" height="2cm"/>
    <svg:text x="5.05cm" y="2.25cm" text-anchor="middle"
      style="font-family:Verdana; font-size:24; font-weight:bold; fill:black">
      Hello, World!
    </svg:text>
    <svg:text x="5cm" y="2.2cm" text-anchor="middle"
      style="font-family:Verdana; font-size:24; font-weight:bold; fill:white">
      Hello, World!
    </svg:text>
  </svg:g>
</svg:svg>
```

When rendered in an SVG viewer, our Hello World document looks like Figure 2-2.

This screen capture was made using the Adobe SVG plug-in inside the Internet Explorer browser. You can find the plug-in at *http://www.adobe.com/svg/*. (Note that many browsers now support SVG natively.)

The Hello World PDF File

To convert the Hello World file into a PDF file, we'll first convert our XML file into formatting objects. The Extensible Stylesheet Language for Formatting Objects (XSL-FO) is an XML vocabulary that describes how content should be rendered. Here is our stylesheet:

```
<?xml version="1.0"?>
<!-- fo-greeting.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format">

  <xsl:output method="xml"/>

  <xsl:template match="/">
    <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
      <fo:layout-master-set>
        <fo:simple-page-master master-name="standard"
          margin-right="75pt" margin-left="75pt"
          page-height="11in" page-width="8.5in"
          margin-bottom="25pt" margin-top="25pt">
          <fo:region-body margin-top="50pt" margin-bottom="50pt"/>
        </fo:simple-page-master>
      </fo:layout-master-set>

      <fo:page-sequence master-reference="standard">
        <fo:flow flow-name="xsl-region-body">
          <xsl:apply-templates select="greeting"/>
        </fo:flow>
      </fo:page-sequence>
    </fo:root>
  </xsl:template>

  <xsl:template match="greeting">
    <fo:block line-height="76pt" font-size="72pt" text-align="center">
      <xsl:value-of select="."/>
    </fo:block>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet converts our Hello World document into the following XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>
    <fo:simple-page-master master-name="standard" margin-right="75pt"
      margin-left="75pt" page-height="11in" page-width="8.5in"
      margin-bottom="25pt" margin-top="25pt">
```

Figure 2-3. PDF version of our Hello World file

```
        <fo:region-body margin-top="50pt" margin-bottom="50pt"/>
      </fo:simple-page-master>
    </fo:layout-master-set>
    <fo:page-sequence master-reference="standard">
      <fo:flow flow-name="xsl-region-body">
        <fo:block line-height="76pt" font-size="72pt" text-align="center">
          Hello, World!
        </fo:block>
      </fo:flow>
    </fo:page-sequence>
  </fo:root>
```

This lengthy set of tags uses formatting objects to describe the size of the page, the margins, font sizes, line heights, etc., along with the text extracted from our XML source document. Now that we have the formatting objects, we can use the Apache XML Project's FOP tool. After converting the formatting objects to PDF, the PDF file looks like Figure 2-3.

Here's the command used to convert our file of formatting objects into a PDF file:

```
java org.apache.fop.apps.Fop greeting.fo greeting.pdf
```

The Hello World Java Program

Our last two transformations don't involve XML vocabularies at all; they use XSLT to convert the Hello World document into other formats. Next, we'll transform our XML source document into the source code for a Java program. When the program is compiled and executed, it prints the message from the XML document to the console. Here's our stylesheet:

```
<?xml version="1.0"?>
<!-- java-greeting.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
  <xsl:output method="text" encoding="UTF-8"/>

  <xsl:template match="/">
    <xsl:text>
public class Greeting
{
  public static void main(String[] argv)
  {
    </xsl:text>
    <xsl:apply-templates select="greeting"/>
    <xsl:text>
  }
}
    </xsl:text>
  </xsl:template>

  <xsl:template match="greeting">
    <xsl:text>System.out.println("</xsl:text>
    <xsl:value-of select="normalize-space()"/>
    <xsl:text>");</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

(Notice that we used `<xsl:output method="text">` to generate text, not markup.) Our stylesheet produces these results:

```
public class Greeting
{
  public static void main(String[] argv)
  {
    System.out.println("Hello, World!");
  }
}
```

The class name defined in the XSLT stylesheet (`Greeting`) must be the name of the generated file. That means we have to specify the case-sensitive filename when we run the transformation. Here's how to do that with Xalan:

```
java org.apache.xalan.xslt.Process -in greeting.xml -xsl java-greeting.xsl
    -out Greeting.java
```

For Saxon, the syntax is slightly simpler:

```
java net.sf.saxon.Transform -o Greeting.java greeting.xml java-greeting.xsl
```

Again, the Schema-aware version of Saxon is slightly different:

```
java.com.saxonica.Transform -o Greeting.java greeting.xml java-greeting.xsl
```

Finally, for MSXSL and the Altova XSLT engine, the commands are:

```
msxsl -o Greeting.java greeting.xml java-greeting.xsl
```

and:

```
altovaxml /xslt1 java-greeting.xsl /in greeting.xml /out Greeting.java
```

When executed, our generated Java program looks like this:

```
C:\> java Greeting
Hello, World!
```

Although generating Java code from an XML document may seem strange, it is actually a very useful technique. The FOP tool from the Apache XML Project does this; it defines a number of properties in XML, then generates the Java source code to create class definitions and `get` and `set` methods for each of those properties.

The Hello World VRML File

For our final transformation, we'll create a VRML file from our XML source document. Here's the stylesheet that does the trick:

```
<?xml version="1.0"?>
<!-- vrml-greeting.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>#VRML V2.0 utf8

Shape
{
  geometry ElevationGrid
  {
    xDimension 9
    zDimension 9
    xSpacing 1
    zSpacing 1
    height
    [
      0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0
    ]
    colorPerVertex FALSE
    color Color
    {
      color
      [
        0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1,
        1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0,
        0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1,
        1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0,
```

```
        0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1,
        1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0,
        0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1,
        1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0, 1 1 1, 0 0 0,
      ]
    }
  }
}

Transform
{
  translation 4.5 1 4
  children
  [
    Shape
    {
      geometry Text
      {
    </xsl:text>
    <xsl:apply-templates select="greeting"/>
    <xsl:text>
        fontStyle FontStyle
        {
          justify "MIDDLE"
          style "BOLD"
        }
      }
    }
  ]
}


NavigationInfo
{
  type ["EXAMINE","ANY"]
}

Viewpoint
{
  position 4 1 10
}
    </xsl:text>
  </xsl:template>

  <xsl:template match="greeting">
    <xsl:text>string"</xsl:text>
    <xsl:value-of select="normalize-space()"/>
    <xsl:text>"</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

As with our earlier stylesheet, our VRML-generating template is mostly boilerplate, with content from the XML source document added at the appropriate point. The `<xsl:apply-templates>` element is replaced with the value of the `<greeting>` element.

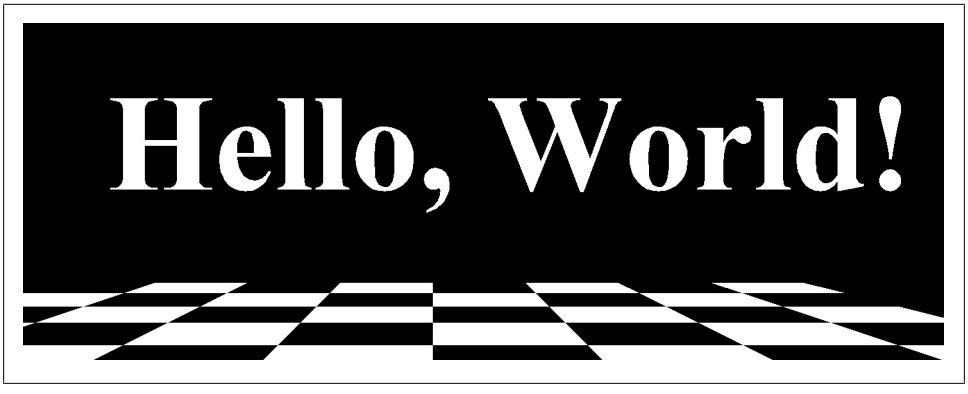


Figure 2-4. One view of the VRML version of our Hello World document

Figure 2-5. Another view of the VRML version of our Hello World document

The VRML code here draws a checkerboard, then draws the text from the XML document above it, floating in midair in the center of the document. A couple of views of the VRML version of our XML document are shown in Figures 2-4 and 2-5.

The screenshots of the VRML world were generated with the Cortona VRML player. You can download the player at *http://www.parallelgraphics.com/cortona*.

Although we haven't discussed any of the specific vocabularies or file formats we've used here, hopefully you understand that you can transform your XML documents into any useful format you can think of. Through the rest of the book, we'll cover several common tasks you can solve with XSLT, all of which build on the basics we've discussed here.

Summary

Although our stylesheets here are trivial, they are much simpler than the corresponding procedural code (written in Visual Basic, C++, Java, etc.) to transform any `<greeting>` elements similarly. We've gone over the basics of what stylesheets are and how they work.

As we go through this book, we'll demonstrate the incredible range of things you can do in XSLT stylesheets, including:

- Using logic, branching, and control statements
- Sorting and grouping elements
- Linking and cross-referencing elements
- Creating master documents that embed other XML documents, then sort, filter, group, and format the combined documents.
- Adding new functions to the XSLT stylesheet processor with XSLT's extension mechanism

XSLT has an extremely active user community. To see just how active, visit the XSL-List site at *http://www.mulberrytech.com/xsl/xsl-list/index.html*.

Before we dive in to those topics, we need to talk about XPath, the syntax that describes what parts of an XML document we want to transform into all of these different things.

CHAPTER 3

XPath: A Syntax for Describing Needles and Haystacks

XPath is a syntax used to describe parts of an XML document. With XPath, you can refer to the first `<para>` element, the `quantity` attribute of the `<part-number>` element, all `<first-name>` elements that contain the text `"Joe"`, and many other variations. In a stylesheet, the XSLT patterns in the `match` and `select` attributes of various elements use XPath syntax to indicate how a document should be transformed. In this chapter, we'll discuss XPath in all its glory.

XPath is designed to be used inside an attribute in an XML document. The syntax is a mix of basic programming language expressions (such as `$x*6`) and Unix-like path expressions (such as `/sonnet/author/last-name`). In addition to the basic syntax, XPath provides a set of useful functions that allow you to find out various things about the document.

One important point, though: XPath works with the parsed version of your XML document. That means that some details of the original document aren't accessible to you from XPath. For example, entity references are resolved by the XSLT processor before instructions in our stylesheet are evaluated. CDATA sections are converted to text as well. That means we have no way of knowing whether a text node in an XPath tree was in the original XML document as text, as an entity reference, or as part of a CDATA section. As you get used to thinking about your XML documents in terms of XPath expressions, this situation won't be a problem, but it may confuse you at first.

[2.0] XPath has undergone enormous changes for version 2.0. Everything that worked in XPath 1.0 still works in XPath 2.0, but there are new capabilities and operators that can greatly simplify your life if you're using an XSLT 2.0 processor. There are three major XPath 2.0 topics that we'll discuss separately in this chapter:

- In XPath 2.0's view of the world, everything is a *sequence*. A sequence replaces the concept of node-sets used in XPath 1.0. The main difference is that a sequence can contain atomic values (more on those in a minute). Be aware that nodes in the sequence (a document node, for example) can still have children. When we're

working with parts of a document, it's common that our sequence is an element node with children and attributes; that one item sequence works just like the trees you know and love from XPath 1.0.

- XPath 2.0 supports *atomic values*. In XPath 1.0, everything from the parsed XML document was a node, including comments, text, and processing instructions. The type system was relatively simple; there were strings, numbers, booleans, node-sets, and something called a result tree fragment. In XPath 2.0, a sequence can contain nodes from an XML document alongside things such as the `xs:integer` value `42` or the `xs:date` value can be part of a sequence along with the nodes used in XPath 1.0. Atomic values are defined in the XML Schema spec; an atomic value is a value that can't be broken down into smaller parts. A value of type `xs:integer` or `xs:date`, for example, is an atomic value.
- XPath 2.0 supports all of the built-in datatypes of XML Schema. This means we can specify that a value must be of a particular datatype, we can create a new value of a particular datatype, and we can cast a given value to a particular datatype. If your XSLT 2.0 processor is schema-aware (more on that shortly), you can define your own datatypes and use them inside XPath along with the XML Schema datatypes. If your XSLT 2.0 processor is not schema-aware, it still supports a number of datatypes from XML Schema.

With those three exceptions, we'll discuss the changes in XPath 2.0 as we cover XPath in general.

The XPath Data Model

XPath 1.0 views an XML document as a tree of nodes. This tree is very similar to a Document Object Model (DOM) tree, so if you're familiar with the DOM, you should have some understanding of how to build basic XPath expressions. (To be precise, this is a conceptual tree; an XSLT processor or anything else that implements the XPath standard doesn't have to build an actual tree.)

[2.0] XPath 2.0 views everything as a sequence. A sequence can contain all of the nodes we cover here as well as atomic values. Whenever we're working with parsed data from an XML document, the sequences we're using are most likely nodes from the document tree, so we won't have to worry about atomic values. For now, just be aware that the underlying data model in XPath 2.0 is different; we'll cover those differences in great detail later in this chapter.

Node Types

There are seven kinds of nodes in XPath:

- The document node (one per document)
- Element nodes

- Attribute nodes
- Text nodes
- Comment nodes
- Processing instruction nodes
- Namespace nodes

We'll talk about all the different node types in terms of the following document:

```
<?xml version="1.0"?>
<?xml-stylesheet href="sonnet.xsl" type="text/xsl"?>

<!DOCTYPE sonnet [
  <!ELEMENT sonnet (auth:author, title, lines)>
  <!ATTLIST sonnet public-domain CDATA "yes"
            type (Shakespearean | Petrarchan) "Shakespearean">
  <!ELEMENT auth:author  (last-name,first-name,nationality,
                          year-of-birth?,year-of-death?)>
  <!ELEMENT last-name (#PCDATA)>
  <!ELEMENT first-name (#PCDATA)>
  <!ELEMENT nationality (#PCDATA)>
  <!ELEMENT year-of-birth (#PCDATA)>
  <!ELEMENT year-of-death (#PCDATA)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT lines (line,line,line,line,
                   line,line,line,line,
                   line,line,line,line,
                   line,line)>
  <!ELEMENT line (#PCDATA)>
]>

<!-- Default sonnet type is Shakespearean, the other allowable  -->
<!-- type is "Petrarchan."                                      -->
<sonnet type='Shakespearean'>
  <auth:author xmlns:auth="http://www.authors.com/">
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
    <year-of-death>1616</year-of-death>
  </auth:author>
  <!-- Is there an official title for this sonnet?  They're
       sometimes named after the first line.                  -->
  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
    <line>If hairs be wires, black wires grow on her head.</line>
    <line>I have seen roses damasked, red and white,</line>
    <line>But no such roses see I in her cheeks.</line>
    <line>And in some perfumes is there more delight</line>
    <line>Than in the breath that from my mistress reeks.</line>
    <line>I love to hear her speak, yet well I know</line>
```

```
    <line>That music hath a far more pleasing sound.</line>
    <line>I grant I never saw a goddess go,</line>
    <line>My mistress when she walks, treads on the ground.</line>
    <line>And yet, by Heaven, I think my love as rare</line>
    <line>As any she belied with false compare.</line>
  </lines>
</sonnet>
<!-- The title of Sting's 1987 album "Nothing like the sun" is  -->
<!-- from line 1 of this sonnet.                                -->
```

The root node

The root node is the XPath node that contains the entire document. In our example, the root node contains the `<sonnet>` element; it's not the `<sonnet>` element itself. In an XPath expression, the root node is specified with a single slash (`/`).

Unlike other nodes, the root node has no parent. It always has at least one child: the document element. The root node also contains comments or processing instructions that are outside the document element. In our sample, the two processing instructions named `xml-stylesheet` and `cocoon-process` are both children of the root node, as are the comments that appear before the `<sonnet>` tag and the comments that appear after it. The string value of the root node (returned by `<xsl:value-of select="/" />`), is the concatenation of all text nodes of the root node's descendants, slammed together without any spaces between them.

Element nodes

Every element in the original XML document is represented by an XPath element node. In the previous document, an element node exists for the `<sonnet>` element, the `<auth:author>` element, the `<last-name>` element, and so on. An element node's children include text nodes, element nodes, comment nodes, and processing instruction nodes that occur within that element in the original document.

An element node's string value (returned by `<xsl:value-of select="sonnet">`, for example) is the concatenation of the text of this node and all of its children, in document order (the order in which they appear in the original document). All entity references (such as `<`) and character references (such as `ß`, the lowercase "sharp S" character) in the text are resolved automatically. To XPath, those characters show up as `<` and `ß`; you can't access the entity or character references from XPath.

The name of an element node (returned by the XPath `name( )` function) is the element name and any namespace in effect. In the previous example, the `name( )` of the `<sonnet>` element is `sonnet`. The `name( )` of the `<auth:author>` element is `auth:author`. Given the name of the node, XPath has functions to return the local name of the element (`author`) and the URI of the namespace associated with the node (`http://www.authors.com`). XPath 2.0 has additional functions to work with qualified names (XML Schema `xs:QName` values) directly, including functions to extract a namespace prefix and the URI associated with it.

Attribute nodes

At a minimum, an element node is the parent of one attribute node for each attribute in the XML source document. In our sample document, the element node corresponding to the `<sonnet>` element is the parent of an attribute node with a name of `type` and a value of `Shakespearean`. A couple of complications for attribute nodes exist, however:

- Although an element node is the parent of its attribute nodes, those attribute nodes are not children of their parent. The children of an element are the text, element, comment, and processing instruction nodes contained in the original element. If you want a document's attributes, you must ask for them specifically. That relationship seems odd at first, but you'll find that treating an element's attributes separately is usually what you want to do.
- If a DTD or schema defines default values for certain attributes, those attributes don't have to appear in the XML document. In our example, we declared an attribute named `public-domain` that has a default value of `yes`. The actual `<sonnet>` element doesn't have this attribute, so the value of its `public-domain` attribute is `yes`. Similarly, the default value for `type` is `Shakespearean`, so a `<sonnet>` element without a `type` attribute uses the default value.

  To make this situation even worse, an XML parser isn't required to read an external DTD. If it doesn't, then any attribute nodes that represent default values not coded in the document won't exist. Fortunately, XSLT has some branching elements (`<xsl:if>` and `<xsl:choose>`) that can help you deal with these ambiguities; we'll discuss those in Chapter 5.
- The XML 1.0 specification defines two attributes (`xml:lang` and `xml:space`) that work differently. In other words, if the `<auth:author>` element in our sample document contains the attribute `xml:lang="en-US"`, that attribute applies to all elements contained inside `<auth:author>`. Even though that attribute might apply to the `<last-name>` element, `<last-name>` won't have an attribute node named `xml:lang`. Similarly, the `xml:space` defines whether whitespace in an element should be preserved; valid values for this attribute are `preserve` and `default`. Whether these attributes are in effect for a given element or not, the only attribute nodes an element node contains are those tagged in the document and those defined with a default value in the DTD.

  We'll discuss handling whitespace in "Dealing with Whitespace" in Chapter 4. For detailed technical information on language codes and whitespace handling, see the discussions of the XPath `lang()` function in Appendix C and also the XSLT `<xsl:preserve-space>` and `<xsl:strip-space>` elements in Appendix A.

Text nodes

Text nodes are refreshingly simple; they contain text from an element. If the original text in the XML document contained entity or character references, they are resolved before the XPath text node is created. The text node is text, pure and simple. A text

node is required to contain as much text as possible; the next or previous node can't be a text node.

You might have noticed that there are no CDATA nodes in this list. If your XML document contains text in a CDATA section, you can access the contents of the CDATA section as a text node. You have no way of knowing if a given text node was originally a CDATA section. Similarly, all entity references are resolved before anything in your stylesheet is evaluated, so you have no way of knowing if a given piece of text originally contained entity references.

Comment nodes

A comment node is also very simple—it has value but no name. Every comment in the source document (except for comments in the DTD) becomes a comment node. The value of the comment node is everything inside the comment—everything between the opening `<!--` and the closing `-->`.

Processing instruction nodes

A processing instruction node has two parts: a name (returned by the `name( )` function) and a string value. The string value is everything after the name, including whitespace, but not including the `?>` that closes the processing instruction.

Namespace nodes

Namespace nodes are almost never used in XSLT stylesheets; they exist primarily for the XSLT processor's benefit. Remember that the declaration of a namespace (such as `xmlns:auth="http://www.authors.net"`), even though it looks like an attribute in the XML source, becomes a namespace node, not an attribute node. Every element in the scope of a namespace declaration has a namespace node. In our sonnet, for example, every node in the `<auth:author>` node has a namespace node.

[2.0] In XPath 2.0, support for namespace nodes is optional.

Node Tests

XPath also has *node tests*. A node test looks like a function, but is used to match certain types of nodes. A node test works like a predicate in that it returns only nodes that meet certain criteria. XPath 1.0 has four node tests:

`node( )`
: Matches all nodes. The test `node( )` is true for every kind of node.

`text( )`
: Matches text nodes only.

`comment( )`
: Matches comment nodes.

`processing-instruction( )`
: Matches processing instruction nodes. If this node test includes a string, it matches processing instruction nodes with that name. For example, `processing-instruction('cocoon-process')` matches processing instruction nodes that begin with `<?cocoon-process>`.

`*`
: Matches all the nodes along a particular axis (we'll cover axes shortly). For example, `child::*` matches all the element children of a node, whereas `attribute::*` matches all the attributes of a node.

`NCName:*`
: Matches all the nodes in a particular namespace. In our sonnet, `auth:*` matches any element that has the namespace URI `http://www.authors.com/`.

[2.0] New node tests in XPath 2.0

XPath 2.0 uses all of the node tests from XPath 1.0. It also defines several new node tests:

`element( )`
: Matches any element. Using this node test without arguments is similar to `select="*"` in XPath 1.0. The difference is that in XPath 2.0, we can use the `element( )` node test to check the name and datatype of an item. For example, `element(author)` matches any elements named `<author>`. The two-argument version of this node test allows us to find elements that match a particular datatype. The node test `element(date-of-birth, xs:gYear)` matches all `<date-of-birth>` elements whose datatype is `xs:gYear`. (That includes datatypes derived from `xs:gYear`.) Finally, we can use a wildcard for the element name to find all of the elements of a particular datatype. `element(*, xs:gYear)` matches all elements with a datatype of `xs:gYear`.

`schema-element(author)`
: Matches any element named `<author>` whose datatype matches the datatype of the `<author>` element declared in a schema. Unlike all the other node tests defined in XPath 1.0 and 2.0, it is an error to use this node test without an argument.

`attribute( )`
: Matches any attribute. Using this node test without arguments is similar to `select="@*"` in XPath 1.0. As with `element( )`, we can use `attribute( )` to check the name and datatype of an item. The node test `attribute(public-domain)` matches any attributes named `public-domain`, regardless of the datatype. `attribute(public-domain, xs:string)` matches only attributes named `public-domain` with a datatype of `xs:string`. Finally, using a wildcard for the *attribute* name returns all the attributes that match a particular datatype. For example, `attribute(*, xs:decimal)` matches all attributes whose datatype is `xs:decimal`, regardless of the name of the attribute.

`*:NCName`
: Matches all the nodes with a particular local name. In our sonnet, `*:author` matches the `<auth:author>` element; it would also match the elements `<xyz:author>` and `<author>`. The match occurs regardless of the namespace URI *or* whether an element has a namespace URI at all.

`document-node( )`
: Matches document nodes. The node test `document-node(element(sonnet))` matches a document node with a single element child (`<sonnet>`). The document node can also contain comments or processing instructions, but it can only contain a single element node.

[2.0] Although `item( )` looks like a node test, it is used only as a datatype. For example, the variable `<xsl:variable name="something" as="item( )">` defines a variable that can contain a single value. The variable can contain any node or atomic type.

[2.0] Sequences and Atomic Values

Working with the XPath 2.0 data model is probably the biggest change in XSLT 2.0. We'll mention the changes to the model as we go along, but we'll discuss three topics outright: sequences, atomic values, and schema support. We'll look at sequences and atomic values now and discuss schema support later in this chapter.

A *sequence* is, well, a sequence of items. Those items might be nodes from an XML document, or they might be simple values such as `'June'` or `3.14`. A sequence has an order and a length. You can use XPath 2.0 functions to see how many items are in a sequence, you can retrieve a subset of the items in the sequence, and you can insert or delete items at a particular point in the sequence. Here is a variable that contains a sequence:

```
<!-- sequences1.xsl -->
...
<xsl:variable name="months" as="xs:string*"
  select="('January', 'February', 'March', 'April',
             'May', 'June', 'July', 'August',
             'September', 'October', 'November',
             'December')"/>
```

There are several things to point out here. First of all, notice that we used the new `as` attribute to define the datatype of this variable. The asterisk here means that the variable can have any number of values. If we defined this variable with the datatype `xs:string`, the variable `$months` could only be a single `xs:string`. The code here would cause a fatal error.

The values in the sequence are all atomic values. An atomic value is a simple value, as opposed to a node. A sequence can contain atomic values, as in our example here; it

can also use XPath to select nodes from an XML document. Here's an example that creates a sequence with atomic values and information from an XML document:

```
<!-- sequences2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="cities" as="xs:string*">
      <xsl:sequence select="addressbook/address/city"/>
      <xsl:sequence select="('London', 'Adelaide', 'Rome')"/>
      <xsl:sequence select="('Jakarta', 'Sao Paulo', 'Timbuktu')"/>
    </xsl:variable>
    <xsl:text>Our customers live in these cities:&#xA;&#xA;</xsl:text>
    <xsl:value-of select="$cities" separator="&#xA;"/>
  </xsl:template>

</xsl:stylesheet>
```

Notice that the variable here contains an `<xsl:sequence>` element to select the `<city>` elements, followed by two `<xsl:sequence>` elements that select three strings each. The value of the sequence `$cities` has 12 items when used with the following document:

```
<!-- names.xml -->
<addressbook>

  <address>
    <name>
      <title>Mr.</title>
      <first-name>Chester Hasbrouck</first-name>
      <last-name>Frisby</last-name>
    </name>
    <street>1234 Main Street</street>
    <city>Sheboygan</city>
    <state>WI</state>
    <zip>48392</zip>
  </address>
...
    <city>Skunk Haven</city>
...
    <city>Winter Harbor</city>
...
    <city>Skunk Haven</city>
...
    <city>Boylston</city>
...
    <city>Lynn</city>
    <state>MA</state>
    <zip>02930</zip>
  </address>
</addressbook>
```

The output from the stylesheet looks like this:

```
Our customers live in these cities:

Sheboygan
Skunk Haven
Winter Harbor
Skunk Haven
Boylston
Lynn
London
Adelaide
Rome
Jakarta
Sao Paulo
Timbuktu
```

The sequence has 12 items: 6 items from the `<city>` elements in the XML document, and 6 values from the `<xsl:sequence>` elements that contain strings. The sequence has a datatype of `xs:string*`, which means *all of the items in the sequence are converted to* `xs:string` *values*. Even though we used the pattern `/addressbook/address/city` to select the nodes, the contents of the sequence are strings. If we change the datatype of the sequence to `item()*`, things work differently:

```
<!-- sequences3.xsl -->
...
    <xsl:variable name="cities" as="item()*">
      <xsl:sequence select="addressbook/address/city"/>
      <xsl:sequence select="('London', 'Adelaide', 'Rome')"/>
      <xsl:sequence select="('Jakarta', 'Sao Paulo', 'Timbuktu')"/>
    </xsl:variable>
    <xsl:for-each select="$cities">
      <xsl:choose>
        <xsl:when test=". instance of element()">
          <xsl:text>        Node: </xsl:text>
        </xsl:when>
        <xsl:otherwise>
          <xsl:text>Atomic value: </xsl:text>
        </xsl:otherwise>
      </xsl:choose>
      <xsl:value-of select="."/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
...
```

This stylesheet now contains 12 items, 6 of which are element nodes, 6 of which are strings. Instead of simply writing the value of each item in the sequence, we use the new `instance of` operator to see whether an item is an element or an atomic value. The results look like this:

```
Our customers live in these cities:

        Node: Sheboygan
        Node: Skunk Haven
        Node: Winter Harbor
```

```
        Node: Skunk Haven
        Node: Boylston
        Node: Lynn
Atomic value: London
Atomic value: Adelaide
Atomic value: Rome
Atomic value: Jakarta
Atomic value: Sao Paulo
Atomic value: Timbuktu
```

```
<xsl:variable name="cities" as="xs:string*">
  <xsl:sequence
    select="(/addressbook/address/city,
             ('London', 'Adelaide', 'Rome'),
             ('Jakarta', 'Sao Paulo', 'Timbuktu'))"/>
</xsl:variable>
```

Location Paths

One of the most common uses of XPath is to create *location paths*. A location path describes the location of something in an XML document. The pattern `/addressbook/address/city` describes the location of the elements we want to select. We'll use location paths as patterns to find parts of the XML document, then we'll use XPath expressions to manipulate them. But before we dive in to the wonders of location paths, we need to discuss the *context*.

The Context

One of the most important concepts in XPath is the context. Everything we do in XPath is interpreted with respect to the context. You can think of an XML document as a hierarchy of directories in a filesystem. In our sonnet example, we could imagine that `sonnet` is a directory at the root level of the filesystem. The `sonnet` directory would, in turn, contain directories named `auth:author`, `title`, and `lines`. In this example, the context would be the current directory. If I go to a command line and execute a particular command (such as `dir *.xsl`), the results I get vary depending on the current directory. Similarly, the results of evaluating an XPath expression will probably vary based on the context.

[1.0] The XPath 1.0 context

Most of the time, we can think of the context as the node in the tree from which any expression is evaluated. To be completely accurate, the context consists of five things:

- The *context node* (the "current directory"). The XPath expression is evaluated from this node.
- Two integers, the *context position* and the *context size*. These integers are important when we're processing a group of nodes. For example, we could write an XPath expression that selects all of the `<li>` elements in a given document. The context

size refers to the number of `<li>` items selected by that expression, and the context position refers to the position of the `<li>` we're currently processing.

- A set of *variables*. This set includes names and values of all variables that are currently in scope.
- A set of all the *functions* available to XPath expressions. Some of these functions are defined by the XPath and XSLT standards themselves; others might be extension functions defined by whomever created the stylesheet. (You'll read more about extension functions in Chapter 9.)
- A set of all the *namespace declarations* currently in scope.

Having said all that, most of the time you can ignore everything but the context node. To use our command-line analogy one more time, if you're at a command line, you have a current directory; you also have (depending on your operating system) a number of environment variables defined. For most commands, you can focus on the current directory and ignore the environment variables.

[2.0] The XPath 2.0 context

As you would expect, the context in XPath 2.0 is more complicated. With support for XML Schema, dates, times, collations, and functions, there are many more things XPath 2.0 has to track.

The specs talk about two things: the *static context* and the *dynamic context*. The static context is the information that's available before an expression is evaluated. That information doesn't change during the evaluation of the expression. The base URI, the default collation, and the variables, namespaces, and schemas in scope are all part of the static context. The context item, the context position, and the context size are all part of the dynamic context. Most of the time we'll be concerned with the dynamic context, but we can access all of the parts of the context whenever we need them.

The static context contains:

- Whether the processor is in *XPath 1.0 compatibility mode*. This is `true` or `false`.
- The set of *statically known namespaces*. Each item in the set is a prefix and a namespace URI.
- The *default namespace for elements and types*. This is either a namespace URI or "none."
- The *default namespace for functions*. This is either a namespace URI or "none."
- The set of *in-scope schema definitions*. These include the schema types, elements, and attributes that are currently in scope.
- The set of *variables* that are in scope.
- The *static type of the context item*.

- The *function signatures* that are currently in scope. These include the namespace and arity (number of parameters) of each function, along with the static types of its parameters and results. Constructor functions are included in this set.
- The *statically known collations*. The format of collations is implementation-defined.
- The *default collation*.
- The *base URI*.
- A set of *statically known documents*. This refers to the documents available through the `doc( )` function. See "[2.0] The doc() and doc-available() Functions" in Chapter 8 for more details on the `doc( )` function.
- A set of *statically known collections*. See the description of the [2.0] `collection( )` function in Appendix C for more information about collections.
- The *statically known default collection type*. The default collection type is `node( )*` unless an XSLT processor has set it to some other value. Again, see the description of the [2.0] `collection( )` function in Appendix C for more information about collections.

The dynamic context contains everything in the static context, plus:

- The *context item*. This is similar to the context node in XPath 1.0, with the difference that the context item can be an atomic value. In XPath 1.0, you could always ask for the parent of the context node. If you try that in XPath 2.0, the processor raises an exception unless the context item is a node.
- The *context position*. This works the same way it did in XPath 1.0; the `position( )` function returns the context position.
- The *context size*. Again, this works the same way it did in XPath 1.0; the `last( )` function returns the context size.
- The set of *variable values* that are in scope.
- The set of *function implementations* in scope.
- The *current dateTime*. This value doesn't change during the execution of an expression.
- The *implicit timezone*. The new [2.0] `implicit-timezone( )` function (see Appendix C) returns this value.
- The set of *available documents* accessible through the [2.0] `doc( )` function (see Appendix C).
- The set of *available collections* accessible through the [2.0] `collection( )` function (see Appendix C).
- The *default collection* returned by calling the [2.0] `collection( )` function without any arguments.

Simple Location Paths

Now that we've talked about what a context is and why it matters, we'll look at some location paths. We'll start with a variety of simple paths; as we go along, we'll look at more complex location paths that use all the various features of XPath. We already looked at one of the simplest XSLT patterns:

```
<xsl:template match="/">
```

This template selects the root node of the document. We saw another simple XPath expression in the `<xsl:value-of>` element:

```
<xsl:value-of select="."/>
```

This template selects the context node, represented by a period. To complete our tour of very simple location paths, we can use the double period (`..`) to select the *parent* of the context node:

```
<xsl:value-of select=".."/>
```

All these XPath expressions have one thing in common: they don't use element names. As you might have noticed in our Hello World example, you can use element names to select elements that have a particular name:

```
<xsl:apply-templates select="greeting"/>
```

In this example, we select all of the `<greeting>` elements in the current context and apply the appropriate template to each of them. Turning to our XML sonnet, we can create location paths that specify more than one level in the document hierarchy:

```
<xsl:apply-templates select="lines/line"/>
```

This example selects all `<line>` elements that are contained in any `<lines>` elements in the current context. If the current context doesn't have any `<lines>` elements, then this expression returns an empty node-set. If the current context has plenty of `<lines>` elements, but none of them contain any `<line>` elements, this expression also returns an empty node-set.

Relative and Absolute Expressions

The XPath specification talks about two kinds of XPath expressions, *relative* and *absolute*. Our previous example is a relative XPath expression because the nodes it specifies depend on the current context. An absolute XPath expression begins with a slash (`/`), which tells the XSLT processor to start at the root of the document, regardless of the current context. In other words, you can evaluate an absolute XPath expression from any context node you want, and the results will be the same. Here's an absolute XPath expression:

```
<xsl:apply-templates select="/sonnet/lines/line"/>
```

The good thing about an absolute expression is that you don't have to worry about the context node. Another benefit is that it makes it easy for the XSLT processor to find all

nodes that match this expression: what we've said in this expression is that there must be a <sonnet> element at the root of the document, that element must contain at least one <lines> element, and that at least one of those <lines> elements must contain at least one <line> element. If any of those conditions fail, the XSLT processor can stop looking through the tree and return an empty node-set.

A disadvantage of using absolute XPath expressions is that it makes your templates more difficult to reuse. Both of these templates process <line> elements, but the second one is more difficult to reuse:

```
<xsl:template match="line">
  ...
</xsl:template>

<xsl:template match="/sonnet/lines/line">
  ...
</xsl:template>
```

If the second template has wonderful code for processing <line> elements, but your document contains <line> elements that don't match the absolute XSLT pattern, you can't reuse that template. (On the other hand, the XSLT processor has less work to do with the pattern /sonnet/lines/line; we've told the processor exactly where to look for the elements we care about.) In general, you should use absolute expressions only if you need special processing for a specific case. If <line> elements that match the pattern /sonnet/lines/line should be processed differently, use the absolute pattern in the match attribute. If you don't need to process those <line>s differently, don't use the absolute pattern. Keep these things in mind as you design your templates.

Selecting Things Besides Elements with Location Paths

Up until now, we've discussed XPath expressions that used either element names (/sonnet/lines/line) or special characters (/ or ..) to select elements from an XML document. Obviously, XML documents contain things other than elements; we'll talk about how to select those other things here.

Selecting attributes

To select an attribute, use the at-sign (@) along with the attribute name. In our sample sonnet, you can select the type attribute of the <sonnet> element with the XPath expression /sonnet/@type. If the context node is the <sonnet> element itself, then the relative XPath expression @type does the same thing.

Selecting the text of an element

To select the text of an element, simply refer to it in your expression. The element <xsl:value-of select="/sonnet/auth:author/last-name"/> returns Shakespeare for our sample sonnet. You can also use the string( ) function, although that's typically not necessary.

These XSLT instructions:

```
<xsl:value-of select="/sonnet/something_else:author/first-name"/>
<xsl:text> </xsl:text>
<xsl:value-of select="/sonnet/something_else:author/last-name/string()"/>
```

generate these results:

```
William Shakespeare
```

Be aware that getting the text of an element with children probably doesn't do what you want. For example, the element `<xsl:value-of select="/sonnet/auth:author"/>` returns the string `ShakespeareWilliamBritish15641616`. All of the text descendants of the `<auth:author>` node are concatenated together. To format these nodes more attractively, you'll have to deal with them individually.

Finally, there is a `text( )` node test that selects the text node children of the context item. That being said, you almost never want to use it. Getting the node's text the way we illustrated here is the best way to go.

Selecting comments, processing instructions, and namespace nodes

By this point, we've covered most of the document components you're ever likely to select with an XPath expression. You can use a couple of other XPath node tests to describe parts of an XML document. The `comment( )` and `processing-instruction( )` node tests allow you to select comments and processing instructions from the XML document. Going back to our sample sonnet, the XPath expression `/processing-instruction( )` returns the two processing instructions (named `xml-stylesheet` and `cocoon-process`). The expression `/sonnet/comment( )` returns the comment node that begins, "Is there an official title for this sonnet?" (That's the only comment in the `<sonnet>` element itself; the other comments are outside the root element.)

Processing comment nodes in this way can actually be useful. If you've entered comments into an XML document, you can use the `comment( )` node test to display your comments only when you want. Here's an XSLT template you could use:

```
<xsl:template match="comment()">
  <span class="comment">
    <p><xsl:value-of select="."/></p>
  </span>
</xsl:template>
```

Elsewhere in your stylesheet, you could define CSS attributes to print comments in a large, bold, purple font. To remove all comments from your output document, simply go to your stylesheet and have the template that handles comments do nothing. (You could also just remove the template; the default template for comments does nothing.)

XPath has one other kind of node—the rarely used *namespace node*. To retrieve namespace nodes, you have to use the *namespace axis*; we'll discuss axes in the "Axes" section later in this chapter. One note about namespace nodes, if you ever have to use them: when matching namespace nodes, the namespace prefix isn't important. As an

example, our sample sonnet used the auth namespace prefix, which maps to the value http://www.authors.com/. If a stylesheet uses the namespace prefix writers to refer to the same URL, then the XPath expression /sonnet/writers:* would return the <auth:author> element. Even though the namespace prefixes are different, the URLs they refer to are the same. Most likely the only time you'll care about the namespace prefix itself is when you're looking for a particular namespace node. The name of the namespace node is the prefix. In almost all cases, you'll use the namespace URI to find what you're looking for.

Having said all that, the chances that you'll ever need to use namespace nodes are pretty slim.

Using Wildcards

XPath features three wildcards:

The asterisk (*)
: Selects all element nodes in the current context. Be aware that the asterisk wildcard selects element nodes only; attributes, text nodes, comments, or processing instructions aren't included. You can also use a namespace prefix with an asterisk. In our sample sonnet, the XPath expression auth:* returns all element nodes in the current context that are associated with the namespace URL http://www.authors.com/.

  [2.0] XPath 2.0 lets us use a wildcard as a namespace prefix. The XPath expression *:author returns all element nodes in the current context that have a local name of author, regardless of their namespace. In XPath 2.0, both auth:* and *:author are legal; in XPath 1.0, looking for *:author causes a fatal error.

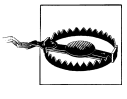
As always, searching for a matching namespace is based on the namespace URL, not the prefix. For example, assume the namespace URL http://www.authors.com/ is associated with the prefix auth in the XML document and the prefix something_else in our stylesheet. Looking for something_else:* in the stylesheet returns all of the elements with the auth prefix in the XML document.

The at-sign and asterisk (@)*
: Selects all attribute nodes in the current context. You can use a namespace prefix with the attribute wildcard. In our sample sonnet, @auth:* returns all attribute nodes in the current context that are associated with the namespace URL http://www.authors.com. (There aren't any attributes associated with the auth namespace; if we added an element such as <first-name auth:nickname="Bill">William</first-name>, the auth:nickname would match this expression.)

[2.0] As with elements, XPath 2.0 lets us use a wildcard for the namespace prefix. The expressions `@auth:*` and `@*:nickname` are both legal in XPath 2.0. In XPath 1.0, using a wildcard for the namespace prefix is a fatal error.

The `node( )` *node test*
: Selects all nodes in the current context, regardless of type. This includes elements, text, comments, processing instructions, attributes, and namespace nodes.

In addition to these wildcards, XPath includes the double slash (`//`), which indicates that zero or more elements may occur between the slashes. For example, the XPath expression `//line` selects all `<line>` elements, regardless of where they appear in the document. This is an absolute XPath expression because it begins with a slash. You can also use the double slash at any point in an XPath expression; the expression `/sonnet/descendant-or-self::node( )/line` selects all `<line>` elements that are descendants of the `<sonnet>` element at the root of the XML document. The expressions `/sonnet//line` and `/sonnet/descendant-or-self::node( )/line` are equivalent. (`descendant-or-self` is an *axis*; we'll talk more about those next.)



The double slash (`//`) is a very powerful operator, but be aware that it can make your stylesheets incredibly inefficient. If we use the XPath expression `//line`, the XSLT processor has to check every node in the document to see whether there are any `<line>` elements. The more specific you can be in your XPath expressions, the less work the XSLT processor has to do and the faster your stylesheets will execute. Thinking back to our filesystem metaphor, if I go to a Windows command prompt and type **`dir/s c:\*.xsl`**, the operating system has to look in every subdirectory for any **.xsl* files that might be there. However, if I type **`dir/s c:\doug\projects\stylesheets\*.xsl`**, the operating system has far fewer places to look, and the command will execute much faster.

Axes

To this point, we've been able to select child elements, attributes, text, comments, and processing instructions with some fairly simple XPath expressions. Obviously, we might want to select many other things, such as:

- All ancestors of the context node
- All descendants of the context node
- All previous siblings or following siblings of the context node (siblings are nodes that have the same parent)

To select these things, XPath provides a number of *axes* (plural of axis) that let you specify various collections of nodes. There are 13 axes in all; we'll discuss all of them here, even though most won't be particularly useful to you. To use an axis in an XPath expression, type the name of the axis, a double colon (`::`), and the name of the element you want to select, if any.

Before we define all of the axes, though, we need to talk about XPath's unabbreviated syntax.

Unabbreviated syntax

To this point, all the XPath expressions we've looked at used the XPath *abbreviated syntax*. Most of the time, that's what you'll use; however, most of the lesser-used axes can only be specified with the unabbreviated syntax. For example, when we wrote an XPath expression to select all of the `<line>` elements in the current context, we used the abbreviated syntax:

```
<xsl:apply-templates select="line"/>
```

If you really enjoy typing, you can use the unabbreviated syntax to specify that you want all of the `<line>` children of the current context:

```
<xsl:apply-templates select="child::line"/>
```

We'll go through all of the axes now, pointing out which ones have an abbreviated syntax.

Axis roll call

The following list contains all of the axes defined by the XPath standard, with a brief description of each:

`child` axis
: Contains the children of the context node. As we've already mentioned, the XPath expression `child::lines/child::line` is equivalent to `lines/line`. If an XPath expression (such as `sonnet`) doesn't have an axis specifier, the `child` axis is used by default. The children of the context node include all comment, element, processing instruction, and text nodes. Attribute and namespace nodes are not considered children of the context node.

`parent` axis
: Contains the parent of the context node, if there is one. (If the context node is the root node, the parent axis returns an empty node-set.) As a step in an XPath expression, the parent axis can be abbreviated with the double period (`..`); this moves up to the current node's parent. If the `<first-name>` and `<last-name>` elements are both children of the `<author>` element and the context node is the `<first-name>` element, the expressions `../last-name`, `parent::author/last-name` and `parent::*/last-name` are equivalent. If the context node does not have a parent, this axis returns an empty node-set.

`self` axis
: Contains the context node itself. As a step in an XPath expression, the `self` axis can be abbreviated with a single period (`.`). The expressions `.`, `self::node( )`, and `self::*` are equivalent in XSLT 1.0.

[2.0] In XSLT 2.0, the self axis selects the context *item*, which might not be a node. If the context item is an atomic value, the expressions `self::node( )` and `self::*` cause the XSLT processor to raise an error. In this case, the only way to access the self axis is with a period. If the context item is a node, the self axis works just as it did in XSLT 1.0.

`attribute` axis
: Contains the attributes of the context node. If the context node is not an element node, this axis is empty. The `attribute` axis can be abbreviated with the at sign (`@`). The expressions `attribute::type` and `@type` are equivalent.

`ancestor` axis
: Contains the parent of the context node, the parent's parent, etc. The `ancestor` axis always contains the root node unless the context node is the root node.

`ancestor-or-self` axis
: Contains the context node, its parent, its parent's parent, and so on. This axis always includes the root node.

`descendant` axis
: Contains all children of the context node, all children of all the children of the context node, and so on. The children are all of the comment, element, processing instruction, and text nodes beneath the context node. In other words, the `descendant` axis does not include attribute or namespace nodes. (As we discussed earlier, although an attribute node has an element node as a parent, an attribute node is not considered a child of that element.)

`descendant-or-self` axis
: Contains the context node and all the children of the context node, all the children of all the children of the context node, all the children of the children of all the children of the context node, and so on. As always, the children of the context node include all comment, element, processing instruction, and text nodes; attribute and namespace nodes are not included.

`preceding-sibling` axis
: Contains all preceding siblings of the context node; in other words, all nodes that have the same parent as the context node and appear before the context node in the XML document. If the context node is an attribute node or a namespace node, the `preceding-sibling` axis is empty.

`following-sibling` axis
: Contains all the following siblings of the context node; in other words, all nodes that have the same parent as the context node and appear after the context node in the XML document. If the context node is an attribute node or a namespace node, the `following-sibling` axis is empty.

`preceding` axis
: Contains all nodes that appear before the context node in the document, except ancestors, attribute nodes, and namespace nodes.

`following` axis
: Contains all nodes that appear after the context node in the document, except descendants, attribute nodes, and namespace nodes.

`namespace` axis
: Contains the namespace nodes of the context node. If the context node is not an element node, this axis is empty.

Predicates

There's one more aspect of XPath expressions that we haven't discussed: *predicates*. These are filters that restrict the nodes selected by an XPath expression. Each predicate is evaluated and converted to a Boolean value (either `true` or `false`). If the predicate is `true` for a given node, that node will be selected; otherwise, it isn't. Predicates always appear inside square brackets (`[]`). Here's an example:

```
<xsl:apply-templates select="line[position() = 7]"/>
```

This expression selects the seventh `<line>` element in the current context. If there are six or fewer `<line>` elements in the current context, this XPath expression returns an empty node-set. Several things can be part of a predicate; we'll go through them here.

Numbers in predicates

Instead of using the `position( )` function, we can use a number. For example, the XPath expression `line[7]` selects the seventh `<line>` element in the context node; this means exactly the same thing as `line[position( ) = 7]`. XPath also provides the boolean `and` and `or` operators as well as the union operator (`|`) to combine predicates. The expression `line[position( )=3 and @style]` matches all `<line>` elements that occur third and that have a `style` attribute, while `line[position( )=3 or @style]` matches all `<line>` elements that either occur third or have a `style` attribute.

You can use more than one predicate if you like; `line[3][@style]` or `line[@style][3]` are both legal. They aren't equivalent, however. Predicates are evaluated from left to right. The XSLT processor handles the first pattern by selecting all of the `<line>` nodes that appear third in a set of sibling `<line>` nodes, then selecting all of those nodes that have a `style` attribute. For the second pattern, the processor selects all the `<line>` elements that have a `style` attribute, then selects the third node from that sequence. The first pattern can match any number of nodes, while the second pattern will never match more than one. In general, the first predicate filters the nodes, the second predicate filters the nodes that made it past the first predicate, and then the third predicate filters the nodes that made it past the second, and so forth.

Functions in predicates

In addition to numbers, we can use XPath and XSLT functions inside predicates. Here are some examples:

`line[last( )]`
: Selects the last `<line>` element in the current context.

`line[position( ) mod 2 = 0]`
: Selects all even-numbered `<line>` elements. (The `mod` operator returns the remainder after a division; the `position` of any even-numbered element divided by 2 has a remainder of 0.)

`sonnet[@type="Shakespearean"]`
: Selects all `<sonnet>` elements that have a `type` attribute with the value `Shakespearean`. Note that double versus single quotes are not significant; this XPath expression matches either `<sonnet type="Shakespearean">` or `<sonnet type='Shakespearean'>`.

`ancestor::table[@border="1"]`
: Selects all `<table>` ancestors of the current context that have a `border` attribute with the value `1`.

`count(/body/table[@border="1"])`
: Returns the number of `<table>` elements with a `border` attribute equal to `1` that are children of `<body>` elements that are children of the root node. Notice that in this case we're using a predicate as part of the location path.

Attribute Value Templates

Although they're technically defined in the XSLT specification (in XSLT 1.0 section 7.6.2 and XSLT 2.0 section 5.6), we'll discuss attribute value templates here. An attribute value template (sometimes abbreviated as AVT) is an XPath expression that is evaluated, and the result of that evaluation replaces the attribute value template. For example, we could create an HTML `<table>` element like this:

```
<table border="{@size}"/>
```

In this example, the XPath expression `@size` is evaluated, and its value, whatever that happens to be, is inserted into the output tree as the value of the `border` attribute. Attribute value templates can be used in any literal result elements in your stylesheet (for HTML elements and other things that aren't part of the XSLT namespace, for example). You can also use attribute value templates in the following XSLT attributes:

- The `name` and `namespace` attributes of the `<xsl:attribute>` element
- The `name` and `namespace` attributes of the `<xsl:element>` element
- The `format`, `lang`, `letter-value`, `grouping-separator`, and `grouping-size` attributes of the `<xsl:number>` element
- The `name` attribute of the `<xsl:processing-instruction>` element
- The `lang`, `data-type`, `order`, and `case-order` attributes of the `<xsl:sort>` element

[2.0] XSLT 2.0 can use AVTs in several additional places:

- The `regex` and `flags` attributes of the new [2.0] `<xsl:analyze-string>` element
- The `name`, `namespace`, and `separator` attributes of the `<xsl:attribute>` element (just as in XSLT 1.0)
- The `name` and `namespace` attributes of the `<xsl:element>` element (just as in XSLT 1.0)
- The `collation` attribute of the new [2.0] `<xsl:for-each-group>` element
- The `terminate` attribute of the `<xsl:message>` element
- The `name` attribute of the [2.0] `<xsl:namespace>` element
- The `format`, `lang`, `letter-value`, `ordinal`, `grouping-separator`, and `grouping-size` attributes of the `<xsl:number>` element (`ordinal` is new in XSLT 2.0; all the others are unchanged from XSLT 1.0)
- The `name` attribute of the `<xsl:processing-instruction>` element
- The `format`, `href`, `method`, `byte-order-mark`, `cdata-section-elements`, `doctype-public`, `doctype-system`, `encoding`, `escape-uri-attributes`, `include-content-type`, `indent`, `media-type`, `normalization-form`, `omit-xml-declaration`, `standalone`, `undeclare-prefixes`, and `output-version` attributes of the new [2.0] `<xsl:result-document>` element
- The `lang`, `order`, `collation`, `stable`, `case-order`, and `data-type` attributes of the `<xsl:sort>` element (`collation` and `stable` are new for XSLT 2.0)
- The `separator` attribute of the `<xsl:value-of>` element (a new attribute in XSLT 2.0)

Datatypes

One of the major additions to XPath 2.0 is support for the XML Schema datatype system. The XPath 1.0 and 2.0 data models are so different we'll discuss them in separate sections. In general, most statements that worked in XPath 1.0 still work in XSLT 2.0. On the other hand, any XPath 2.0 statement that uses the new datatyping features won't work at all in XPath 1.0.

Datatypes in XPath 1.0

In XPath 1.0, an expression returns one of four datatypes:

`node-set`
: Represents a set of nodes. The set can be empty or it can contain any number of nodes.

`boolean`
: Represents the value `true` or `false`. Be aware that the `true` or `false` strings have no special meaning or value in XPath; see "Converting to boolean values" in Chapter 5 for a more detailed discussion of these.

`number`
: Represents a floating-point number. All numbers in XPath and XSLT are implemented as floating-point numbers; the `integer` (or `int`) datatype does not exist in XPath and XSLT. Specifically, all numbers are implemented as IEEE 754 floating-point numbers, which is the same standard used by the Java `float` and `double` primitive types. In addition to ordinary numbers, there are five special values for numbers: positive and negative infinity, positive and negative zero, and `NaN`, the special symbol for anything that is not a number.

`string`
: Represents zero or more characters, as defined in the XML specification.

These datatypes are usually simple, and with the exception of node-sets, converting between types is usually straightforward. We won't discuss these datatypes in any more detail here; instead, we'll discuss datatypes and conversions as we need them to do specific tasks.

Datatypes in XPath 2.0

The XPath 2.0 data model is perhaps the most significant change to writing XSLT version 2.0 stylesheets. We'll cover the datatypes supported by XPath 2.0. XPath 2.0 supports all of the basic datatypes defined in XML Schema, and a schema-aware XSLT 2.0 processor lets you create your own datatypes. We'll start with the basic datatypes; *these are the only datatypes supported by a basic XSLT processor*. To support other datatypes, including datatypes we define (`po:purchaseOrder`, for example) and derived types defined in XML Schema (such as `xs:nonNegativeInteger`), you need a schema-aware XSLT processor.

We've already looked at using `<xsl:variable name="sample" select="'3'" as="xs:integer"/>` as a way of creating an `xs:integer` value. XPath 2.0 also provides *constructor functions*, described in the following list. For example, `<xsl:variable name="sample" select="xs:integer(3)"/>` creates a new `xs:integer` value, whereas `<xsl:variable name="birthday" select="xs:date('1995-04-21')"/>` creates a new `xs:date` value:

`xs:string`
: The `xs:string` datatype represents a string. Every datatype supported by XPath 2.0 has a *string representation*. If you want to see how a datatype looks as a string, the XSLT `<xsl:value-of>` element will do the trick. You can convert anything to a string by using the constructor function `xs:string( )`. The constructor is the equivalent of the Java `toString( )` method that's inherited by every class.

`xs:boolean`
: As in XSLT 1.0, the string values `true` and `false` don't have any special meaning. To work with the boolean values themselves, XPath provides the `true( )` and `false( )` functions; they return the corresponding boolean values. See "Converting

to boolean values" in Chapter 5 for a more detailed discussion of these values. You can also use the `xs:boolean( )` constructor to create boolean values. `xs:boolean(1)` creates the value `true`, while `xs:boolean(0)` creates the value `false`.

`xs:decimal`
: XML Schema defines an `xs:decimal` value as a numeric value consisting of decimal digits (`0 through 9`), beginning with an optional plus or minus sign. *An* `xs:decimal` *cannot contain an exponent*. The XML Schema spec states that an implementation must support a minimum of 18 decimal digits. The values `42`, `8.37284`, and `-83982.22` are all legal `xs:decimal` values.

`xs:float` *and* `xs:double`
: The `xs:float` and `xs:double` datatypes are based on the IEEE single-precision and double-precision floating-point types, respectively. Unlike `xs:decimal`, `xs:float` and `xs:double` values can have exponents. There are three special values of `xs:float` and `xs:double`: `INF` (infinity), `-INF` (negative infinity), and `NaN` (not a number). The values `42`, `8.37284`, `-83982.22`, `-8.39822e4`, `INF`, and `-0` are all valid values for an `xs:float` or `xs:double`.

`xs:integer`
: An integer is a number without a decimal point or digits after it. An integer can include a plus or minus sign (`+` or `-`) to indicate a negative value; without either, the integer is assumed to be positive.

`xs:duration`
: An `xs:duration` represents a span of time. It has six components: year, month, day, hour, minute, and second. The XML Schema spec states that an implementation should support at least a four-digit year and a seconds value with at least three decimal points (millisecond precision). A duration of 1 year, 7 months, 18 days, 4 hours, 27 minutes, and 3.673 seconds is written as `P1Y7M18DT4H27M3.673S`.

Date and time values
: There are three datatypes for dates and times: `xs:date`, `xs:time`, and `xs:dateTime`. The format of a date is `YYYY-MM-DD`, as in `1995-04-21` for April 21st, 1995. A time value is in the format `hh:mm:ss.sss`, so `17:38:22.183` is the same as 22.183 seconds past 5:38 p.m.

  Both `xs:date` and `xs:time` values have an optional time zone indicator, shown by a plus or minus sign (`+ | -`) that indicates that the date or time is some number of hours ahead or behind Coordinated Universal Time (UTC, also known as Greenwich Mean Time). For example, during the winter (when Daylight Savings Time is not in effect), the time zone on the East coast of the United States is `-05:00`. Be aware that if the XSLT processor normalizes a date or time value, parts of the value can change. The time value `17:30:22.183-05:00` is the same as the time value `00:30:22.183Z` (a date or time value that ends with `Z` has been normalized to UTC).

  Finally, an `xs:dateTime` value is the combination of an `xs:date` and an `xs:time`. The written representation of an `xs:dateTime` has a `T` between the two portions. To combine our earlier examples, `1995-04-21T17:38:22.183-05:00` is 22.183 seconds

past 5:38 p.m. on April 21st, 1995, five hours behind UTC. That value is equivalent to `1995-04-22T00:38:22.183Z`.

Be aware that `xs:date`, `xs:time`, and `xs:dateTime` can have negative values.

Parts of date and time values
: XML Schema defines the datatypes `xs:gYearMonth`, `xs:gYear`, `xs:gMonthDay`, `gDay`, and `gMonth`. Examples of these values, in order, are `1995-04` for April, 1995; `1995` for the year 1995; `--04-21` for the 21st day of April; `---21` for the 21st day of a month; and `--04` for April.

`xs:hexBinary` *and* `xs:base64Binary`
: The `xs:hexBinary` datatype is a string composed of binary octets. In other words, it must contain only pairs of hexadecimal digits ([0-9a-fA-F]). The `xs:base64Binary` datatype uses base 64 encoding to represent binary data. It is also a string. An `xs:base64Binary` value consists of the 65 characters defined in RFC2045: [0-9a-zA-Z], the plus sign (`+`), the forward slash (`/`), and the equals sign (`=`), along with whitespace characters. (The RFC is available at *http://www.ietf.org/rfc/rfc2045.txt*.)

`xs:anyURI`
: An `xs:anyURI` value is any string that forms a valid URI as defined by RFC 2396 (and later updated by RFC2732). The RFCs are available at *http://www.ietf.org/rfc/rfc2396.txt* and *http://www.ietf.org/rfc/rfc2732.txt*.

`xs:QName`
: An `xs:QName` (qualified name) is an XML name qualified with a namespace prefix. For example, `auth:author` is a qualified name. To use an `xs:QName` in a stylesheet, the namespace prefix must be in scope.

`xs:anyType` *and* `xs:anySimpleType`
: The datatype `xs:anyType` is considered the base datatype (called the *ur-type* in the XML Schema datatypes spec) from which all other datatypes are derived. A value of `xs:anyType` can contain any data; it is not constrained in any way.

  The `xs:anySimpleType` datatype is a restricted version of `xs:anyType`; an `xs:anySimpleType` value can be any legal value for any of the primitive datatypes defined in XML Schema. A *primitive datatype* is a datatype that is not defined in terms of another. The `xs:float` datatype is a primitive datatype, so an `xs:float` value would be considered `xs:anySimpleType`. On the other hand, `xs:integer` is not a simple type; it is defined in terms of `xs:float`.

The following datatypes were added by the XPath 2.0 and XQuery 1.0 Data Model spec. They are supported along with all of the datatypes defined by XML Schema:

`xs:yearMonthDuration` *and* `xs:dayTimeDuration`
: These two types were added to the XML Schema datatypes namespace by the XPath 2.0 and XQuery 1.0 Data Model spec. They represent the two halves of an `xs:duration`. An `xs:yearMonthDuration` is some number of years, months, and days, whereas an `xs:dayTimeDuration` is some number of days, hours, minutes, and

seconds. Both of these datatypes can have negative values. The duration 12 years and 2 months is written as `P12Y2M`. Note that an `xs:yearMonthDuration` doesn't have a days component. The duration 4 days, 7 hours, 47 minutes, and 32.883 seconds is written as `P4DT7H47M32.883S`.

`xs:untyped` *and* `xs:untypedAtomic`
: These datatypes are defined by the XPath 2.0 and XQuery 1.0 Data Model spec. A node that has not been validated has a dynamic type of `xs:untyped`, whereas an atomic value that has not been validated has a dynamic type of `xs:untypedAtomic`.

`xs:anyAtomicType`
: In XPath 2.0 and XQuery 1.0, all simple types have `xs:anyAtomicType` as their base type. For example, `xs:integer`, `xs:boolean`, and `xs:string` are all derived from `xs:anyAtomicType`.

XPath Operators

XPath supports a number of operators that make your expressions more powerful. We'll look at each of them here.

[1.0] The first two sections here cover all of the XPath 1.0 operators except the vertical bar (|), the union operator. Only the vertical bar is supported in XPath 1.0; the new `union` keyword is supported only in XPath 2.0. See the section "[2.0] Set Operators—except, intersect, and union" later in this chapter for the details of the union operator (|) and the `union` keyword.

[2.0] In XPath 2.0, some operators work with dates, times, and durations. For example, you can add 12 hours to an `xs:dayTimeDuration` if you want. We cover the operators and features specific to XPath 2.0 after looking at the features common to XPath 1.0 and 2.0.

Mathematical Operators

The mathematical operators available in XPath are pretty limited. We'll use two stylesheets to illustrate how the operators work; the first indicates how an operator works in XSLT 1.0 and the second indicates how it works in XSLT 2.0. We'll cover the stylesheets in detail for the first operator (the plus sign), then simply refer to those examples throughout this section.

Addition (+)

The plus sign adds two numbers.

[1.0] In an XSLT 1.0 stylesheet, the processor attempts to convert each operand to a number. The following XPath expressions all work in XPath 1.0:

```
<?xml version="1.0"?>
<!-- addition-1_0.xsl -->
```

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Tests of addition in XPath 1.0&#xA;</xsl:text>
    <xsl:text>&#xA;  9 + 3 = </xsl:text>
    <xsl:value-of select="9 + 3"/>
    <xsl:text>&#xA;  9 + 3.8 = </xsl:text>
    <xsl:value-of select="9 + 3.8"/>
    <xsl:text>&#xA;  9 + '4' = </xsl:text>
    <xsl:value-of select="9 + '4'"/>
    <xsl:text>&#xA;  9 + 'Q' = </xsl:text>
    <xsl:value-of select="9 + 'Q'"/>
    <xsl:text>&#xA;  9 + true() = </xsl:text>
    <xsl:value-of select="9 + true()"/>
    <xsl:text>&#xA;  9 + false() = </xsl:text>
    <xsl:value-of select="9 + false()"/>
  </xsl:template>
</xsl:stylesheet>
```

Here are the stylesheet results:

```
Tests of addition in XPath 1.0

  9 + 3 = 12
  9 + 3.8 = 12.8
  9 + '4' = 13
  9 + 'Q' = NaN
  9 + true() = 10
  9 + false() = 9
```

Notice that XSLT 1.0 converts a string (`'4'`) to a number. If the string can't be converted to a number (`'Q'`), the result is `NaN`, or not a number. The boolean values returned by the functions `true( )` and `false( )` are converted to the numbers `1` and `0`, respectively.

[2.0] XSLT 2.0 is much more strict. The two operands must be compatible datatypes. Changing the stylesheet to `<xsl:stylesheet version="2.0" ...>` causes the stylesheet to fail. The first two operations (`9 + 3` and `9 + 3.8`) work, but none of the others do. To fix the problem, we can use the `number( )` function to convert the values to numbers. The updated stylesheet looks like this:

```
<?xml version="1.0"?>
<!-- addition-2_0.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Tests of addition in XPath 2.0&#xA;</xsl:text>
    <xsl:text>&#xA;  9 + 3 = </xsl:text>
```

```
    <xsl:value-of select="9 + 3"/>
    <xsl:text>&#xA;  9 + 3.8 = </xsl:text>
    <xsl:value-of select="9 + 3.8"/>
    <xsl:text>&#xA;  9 + number('4') = </xsl:text>
    <xsl:value-of select="9 + number('4')"/>
    <xsl:text>&#xA;  9 + number('Q') = </xsl:text>
    <xsl:value-of select="9 + number('Q')"/>
    <xsl:text>&#xA;  9 + number(true()) = </xsl:text>
    <xsl:value-of select="9 + number(true())"/>
    <xsl:text>&#xA; 9 + number(false()) = </xsl:text>
    <xsl:value-of select="9 + number(false())"/>
  </xsl:template>
</xsl:stylesheet>
```

Now that we explicitly cast each value to a number, the stylesheet generates the same results as the XSLT 1.0 stylesheet:

```
Tests of addition in XPath 2.0

  9 + 3 = 12
  9 + 3.8 = 12.8
  9 + number('4') = 13
  9 + number('Q') = NaN
  9 + number(true()) = 10
  9 + number(false()) = 9
```

A final option is to add the XSLT 2.0 attribute `version="1.0"` to any of the `<xsl:value-of>` elements. This works in an XSLT 2.0 stylesheet:

```
<xsl:value-of select="9 + '4'" version="1.0"/>
```

This generates the value `13`. The `version` attribute can be added to any XSLT element to specify that it should be processed as if it were in an XSLT 1.0 stylesheet.

If the values we're adding are nodes instead of atomic values, the dynamic typing from XPath 1.0 works the same way. For example, we can use this subtraction to find Shakespeare's age:

```
<xsl:value-of select="/sonnet/auth:author/year-of-death -
                      /sonnet/auth:author/year-of-birth"/>
```

We're subtracting two nodes here, so both of them are converted to numbers and subtracted. Trying to do the same thing with two strings doesn't work in XSLT 2.0:

```
<xsl:value-of select="'1616' - '1564'"/>
```

[2.0] In XPath 2.0, the plus sign can be used to add combinations of dates, times, and durations. You can use the plus sign to add the following:

- Two `xs:yearMonthDuration`s
- Two `xs:dayTimeDuration`s
- An `xs:yearMonthDuration` to an `xs:dateTime`

- An `xs:dayTimeDuration` to an `xs:dateTime`
- An `xs:yearMonthDuration` to an `xs:date`
- An `xs:dayTimeDuration` to an `xs:date`
- An `xs:dayTimeDuration` to an `xs:time`.

(Notice that you can't add an `xs:yearMonthDuration` to an `xs:time`.)

Here is a stylesheet with examples of all the supported types of addition:

```
<?xml version="1.0"?>
<!-- addition-datesTimesDurations.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:variable name="yMD1" as="xs:yearMonthDuration"
    select="xs:yearMonthDuration('P1Y8M')"/>
  <xsl:variable name="yMD2" as="xs:yearMonthDuration"
    select="xs:yearMonthDuration('P2Y7M')"/>
  <xsl:variable name="dTD1" as="xs:dayTimeDuration"
    select="xs:dayTimeDuration('P5DT9H23M12S')"/>
  <xsl:variable name="dTD2" as="xs:dayTimeDuration"
    select="xs:dayTimeDuration('P3DT16H12M17S')"/>
  <xsl:variable name="dT" as="xs:dateTime"
    select="xs:dateTime('1995-04-21T00:47:00')"/>
  <xsl:variable name="d" as="xs:date"
    select="xs:date('1995-04-21')"/>
  <xsl:variable name="t" as="xs:time"
    select="xs:time('17:03:00')"/>

  <xsl:template match="/">
    <xsl:text>More tests of addition in XPath 2.0:</xsl:text>
    <xsl:text>&#xA;&#xA;  Two xs:yearMonthDurations:&#xA;    </xsl:text>
    <xsl:value-of select="($yMD1, '+', $yMD2, '=', $yMD1 + $yMD2)"/>
    <xsl:text>&#xA;&#xA;  Two xs:dayTimeDurations:&#xA;    </xsl:text>
    <xsl:value-of select="($dTD1, '+', $dTD2, '=', $dTD1 + $dTD2)"/>
    <xsl:text>&#xA;&#xA;  An xs:yearMonthDuration and an </xsl:text>
    <xsl:text>xs:dateTime:&#xA;    </xsl:text>
    <xsl:value-of select="($dT, '+', $yMD1, '=', $dT + $yMD1)"/>
    <xsl:text>&#xA;&#xA;  An xs:dayTimeDuration and an </xsl:text>
    <xsl:text>xs:dateTime:&#xA;    </xsl:text>
    <xsl:value-of select="($dT, '+', $dTD1, '=', $dT + $dTD1)"/>
    <xsl:text>&#xA;&#xA;  An xs:yearMonthDuration and an </xsl:text>
    <xsl:text>xs:date:&#xA;    </xsl:text>
    <xsl:value-of select="($d, '+', $yMD1, '=', $d + $yMD1)"/>
    <xsl:text>&#xA;&#xA;  An xs:dayTimeDuration and an </xsl:text>
    <xsl:text>xs:date:&#xA;    </xsl:text>
    <xsl:value-of select="($d, '+', $dTD1, '=', $d + $dTD1)"/>
    <xsl:text>&#xA;&#xA;  An xs:dayTimeDuration and an </xsl:text>
    <xsl:text>xs:time:&#xA;    </xsl:text>
    <xsl:value-of select="($t, '+', $dTD1, '=', $t + $dTD1)"/>
```

```
  </xsl:template>
</xsl:stylesheet>
```

Here are the results:

```
More tests of addition in XPath 2.0:

  Two xs:yearMonthDurations:
    P1Y8M + P2Y7M = P4Y3M

  Two xs:dayTimeDurations:
    P5DT9H23M12S + P3DT16H12M17S = P9DT1H35M29S

  An xs:yearMonthDuration and an xs:dateTime:
    1995-04-21T00:47:00 + P1Y8M = 1996-12-21T00:47:00

  An xs:dayTimeDuration and an xs:dateTime:
    1995-04-21T00:47:00 + P5DT9H23M12S = 1995-04-26T10:10:12

  An xs:yearMonthDuration and an xs:date:
    1995-04-21 + P1Y8M = 1996-12-21

  An xs:dayTimeDuration and an xs:date:
    1995-04-21 + P5DT9H23M12S = 1995-04-26

  An xs:dayTimeDuration and an xs:time:
    17:03:00 + P5DT9H23M12S = 02:26:12
```

Subtraction (–)

Given two numbers, the minus sign subtracts the second number from the first. Using subtraction in our sample stylesheets, we get these results:

```
Tests of XPath subtraction in XSLT 1.0

  9 - 3 = 6
  9 - 3.8 = 5.2
  9 - '4' = 5
  9 - 'Q' = NaN
  9 - true() = 8
  9 - false() = 9
```

We get the same results in an XSLT 2.0 stylesheet by casting each of the arguments to numbers or by using the `version` attribute.

[2.0] In XPath 2.0, the minus sign can be used to subtract durations from each other and from `xs:date`, `xs:dateTime`, and `xs:time` values. We can use the minus sign to subtract:

- One `xs:yearMonthDuration` from another
- One `xs:dayTimeDuration` from another
- An `xs:yearMonthDuration` from an `xs:dateTime`
- An `xs:dayTimeDuration` from an `xs:dateTime`

- An `xs:yearMonthDuration` from an `xs:date`
- An `xs:dayTimeDuration` from an `xs:date`
- An `xs:dayTimeDuration` from an `xs:time`.

Replacing the plus signs with minus signs in our previous stylesheet gives us these results:

```
More tests of subtraction in XPath 2.0:

  One xs:yearMonthDuration from another:
    P1Y8M - P2Y7M = -P11M

  One xs:dayTimeDuration from another:
    P5DT9H23M12S - P3DT16H12M17S = P1DT17H10M55S

  An xs:yearMonthDuration from an xs:dateTime:
    1995-04-21T00:47:00 - P1Y8M = 1993-08-21T00:47:00

  An xs:yearMonthDuration from an xs:dateTime:
    1995-04-21T00:47:00 - P1Y8M = 1993-08-21T00:47:00

  An xs:dayTimeDuration from an xs:dateTime:
    1995-04-21T00:47:00 - P5DT9H23M12S = 1995-04-15T15:23:48

  An xs:yearMonthDuration from an xs:date:
    1995-04-21 - P1Y8M = 1993-08-21

  An xs:dayTimeDuration from an xs:date:
    1995-04-21 - P5DT9H23M12S = 1995-04-15

  An xs:dayTimeDuration from an xs:time:
    17:03:00 - P5DT9H23M12S = 07:39:48
```

Notice that the first subtraction returns the *negative* value `-P11M`.

Multiplication (*)

Given two numbers, the multiplication sign multiplies the first number by the second. Again, using the * operator in our stylesheets gives us these results:

```
Tests of XPath multiplication in XPath 1.0

  9 * 3 = 27
  9 * 3.8 = 34.199999999999996
  9 * '4' = 36
  9 * 'Q' = NaN
  9 * true() = 9
  9 * false() = 0
```

The results in XPath 2.0 are cleaner; `34.19999...` is rounded to `34.2`:

```
Tests of XPath multiplication in XPath 2.0

  9 * 3 = 27
  9 * 3.8 = 34.2
```

```
9 * number('4') = 36
9 * number('Q') = NaN
9 * number(true()) = 9
9 * number(false()) = 0
```

[2.0] In XPath 2.0, you can multiply `xs:yearMonthDuration`s and `xs:dayTimeDuration`s by numeric values:

```
<?xml version="1.0"?>
<!-- multiplication-datesTimesDurations.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:variable name="yMD1" as="xs:yearMonthDuration"
    select="xs:yearMonthDuration('P1Y8M')"/>
  <xsl:variable name="dTD1" as="xs:dayTimeDuration"
    select="xs:dayTimeDuration('P24DT8H0M0S')"/>

  <xsl:template match="/">
    <xsl:text>More tests of multiplication in XPath 2.0:</xsl:text>
    <xsl:text>&#xA;&#xA;  A xs:yearMonthDuration multiplied </xsl:text>
    <xsl:text>by a number:&#xA;    </xsl:text>
    <xsl:value-of select="($yMD1, '* 3 =', $yMD1 * 3)"/>
    <xsl:text>&#xA;&#xA;  A xs:dayTimeDuration multiplied </xsl:text>
    <xsl:text>by a number:&#xA;    </xsl:text>
    <xsl:value-of select="($dTD1, '* 10.5 =', $dTD1 * 10.5)"/>
  </xsl:template>
</xsl:stylesheet>
```

The results look like this:

```
More tests of multiplication in XPath 2.0:

  A xs:yearMonthDuration multiplied by a number:
    P1Y8M * 3 = P5Y

  A xs:dayTimeDuration multiplied by a number:
    P24DT8H * 10.5 = P255DT12H
```

(Technically, the numeric value should be an `xs:double`, but XSLT 2.0 converts the numeric value `3` automatically.)

Division (div)

Given two numbers, the division sign divides the first number by the second. In most programming languages, the division operator is the slash (/), but XPath uses the slash as a separator in location paths, so we use the more verbose `div` operator instead. Using `div` in our stylesheets, here are the results for XPath 1.0:

```
Tests of XPath div in XPath 1.0

  9 div 3 = 3
```

```
9 div 3.8 = 2.368421052631579
9 div '4' = 2.25
9 div 'Q' = NaN
9 div true() = 9
9 div false() = INF
```

The results for XPath 2.0 are very similar:

```
Tests of XPath div in XPath 2.0

9 div 3 = 3
9 div 3.8 = 2.368421052631578947
9 div number('4') = 2.25
9 div number('Q') = NaN
9 div number(true()) = 9
9 div number(false()) = INF
```

Notice that the value `9 div false( )`, equivalent to `9 div 0`, returns the value `INF` (infinity). Dividing by zero is not a fatal error, as you might expect.

[2.0] You can also divide durations in four different ways:

- Divide an `xs:yearMonthDuration` by an `xs:double`
- Divide one `xs:yearMonthDuration` by another
- Divide an `xs:dayTimeDuration` by an `xs:double`
- Divide one `xs:dayTimeDuration` by another

We'll use this stylesheet:

```
<?xml version="1.0"?>
<!-- div-datesTimesDurations.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:variable name="yMD1" as="xs:yearMonthDuration"
    select="xs:yearMonthDuration('P1Y8M')"/>
  <xsl:variable name="yMD2" as="xs:yearMonthDuration"
    select="xs:yearMonthDuration('P0Y5M')"/>
  <xsl:variable name="dTD1" as="xs:dayTimeDuration"
    select="xs:dayTimeDuration('P24DT8H0M0S')"/>
  <xsl:variable name="dTD2" as="xs:dayTimeDuration"
    select="xs:dayTimeDuration('P0DT4H0M0S')"/>

  <xsl:template match="/">
    <xsl:text>More tests of division in XPath 2.0:</xsl:text>
    <xsl:text>&#xA;&#xA;  A xs:yearMonthDuration divided </xsl:text>
    <xsl:text>by a number:&#xA;    </xsl:text>
    <xsl:value-of select="($yMD1, 'div 4 =', $yMD1 div 4)"/>
    <xsl:text>&#xA;&#xA;  One xs:yearMonthDuration divided </xsl:text>
    <xsl:text>by another:&#xA;    </xsl:text>
    <xsl:value-of select="($yMD1, 'div', $yMD2, '=', $yMD1 div $yMD2)"/>
    <xsl:text>&#xA;&#xA;  A xs:dayTimeDuration divided </xsl:text>
```

```
    <xsl:text>by a number:&#xA;    </xsl:text>
    <xsl:value-of select="($dTD1, 'div 4.5 =', $dTD1 div 4.5)"/>
    <xsl:text>&#xA;&#xA;  One xs:dayTimeDuration divided </xsl:text>
    <xsl:text>by another:&#xA;    </xsl:text>
    <xsl:value-of select="($dTD1, 'div', $dTD2, '=', $dTD1 div $dTD2)"/>
  </xsl:template>
</xsl:stylesheet>
```

The results look like this:

```
More tests of division in XPath 2.0:

  A xs:yearMonthDuration divided by a number:
    P1Y8M div 4 = P5M

  One xs:yearMonthDuration divided by another:
    P1Y8M div P5M = 4

  A xs:dayTimeDuration divided by a number:
    P24DT8H div 4.5 = P5DT9H46M40S

  One xs:dayTimeDuration divided by another:
    P24DT8H div PT4H = 146
```

[2.0] Integer division (idiv)

XPath 2.0 introduces the `idiv` operator for integer division. The rules for integer division in XPath 2.0 are different from the rules you might know from C++ and Java; no rounding is done if there is any remainder from the division. Changing our earlier `div` example to use `idiv`, we have to remove a couple of error cases:

```
<?xml version="1.0"?>
<!-- idiv.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Tests of idiv in XPath 2.0&#xA;</xsl:text>
    <xsl:text>&#xA;  9 idiv 3 = </xsl:text>
    <xsl:value-of select="9 idiv 3"/>
    <xsl:text>&#xA;  9 idiv 3.8 = </xsl:text>
    <xsl:value-of select="9 idiv 3.8"/>
    <xsl:text>&#xA;  9 idiv number('4') = </xsl:text>
    <xsl:value-of select="9 idiv number('4')"/>
    <!-- Causes a fatal error -->
    <!-- <xsl:value-of select="9 idiv number('Q')"/> -->
    <xsl:text>&#xA;  9 idiv number(true()) = </xsl:text>
    <xsl:value-of select="9 idiv number(true())"/>
    <!-- Causes a fatal error -->
    <!-- <xsl:value-of select="9 idiv number(false())"/> -->
  </xsl:template>
</xsl:stylesheet>
```

Compare these results to those for the `div` operator:

```
Tests of idiv in XPath 2.0

  9 idiv 3 = 3
  9 idiv 3.8 = 2
  9 idiv number('4') = 2
  9 idiv number(true()) = 9
```

As you can see, no rounding is done.

Modulo (mod)

The `mod` operator returns the remainder of a division. Here are the results for XPath 1.0:

```
Tests of the mod operator in XPath 1.0

  9 mod 3 = 0
  9 mod 3.8 = 1.4000000000000004
  9 mod '4' = 1
  9 mod 'Q' = NaN
  9 mod true() = 0
  9 mod false() = NaN
```

The results for XPath 2.0 are very similar:

```
Tests of the mod operator in XPath 2.0

  9 mod 3 = 0
  9 mod 3.8 = 1.4
  9 mod number('4') = 1
  9 mod number('Q') = NaN
  9 mod number(true()) = 0
  9 mod number(false()) = NaN
```

When we use `mod` to divide by zero, we get the value `NaN` (not a number). This is the same result we get when we divide by something that can't be converted to a number (`'Q'`, for example).

The most common use of the `mod` operator is to cycle through a set of values. For example, say we want a stylesheet that generates different background colors for rows of a table. We want to alternate between white and gray backgrounds. We can use the `mod` operator for this:

```
<xsl:attribute name="bgcolor">
  <xsl:choose>
    <xsl:when test="position() mod 2 = 1">
      <xsl:text>white</xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:text>gray</xsl:text>
    </xsl:otherwise>
  </xsl:choose>
</xsl:attribute>
```

The position of the first item is 1. The remainder of dividing 1 by 2 is 1, so the first row will have a white background. For every alternate row, the remainder will be 0, which means the background color will be gray. To alternate between five different values, you would write tests such as `position( ) mod 5 = 1`, `position( ) mod 5 = 2`, and so on.

Unary minus (–x)

The unary minus sign returns the negation of its operand. Here is a stylesheet that illustrates how it works:

```
<?xml version="1.0"?>
<!-- unary-minus.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="x" as="xs:integer" select="xs:integer(-10)"/>

    <xsl:text>An example of the unary minus </xsl:text>
    <xsl:text>operator:&#xA;    </xsl:text>
    <xsl:value-of select="('$x =', $x, '&#xA;   -$x = ', -$x)"/>
  </xsl:template>

</xsl:stylesheet>
```

The results look like this:

```
An example of the unary minus operator:
    $x = -10
   -$x =  10
```

Some details from the Functions and Operators spec: if the argument is an `xs:integer` or `xs:decimal`, the negation of `0` or `0.0` is `0` and `0.0`, respectively. If the argument is an `xs:float` or `xs:double`, `NaN` returns `NaN`, `0.0E0` returns `-0.0E0`, `-0.0E0` returns `0.0E0`, `INF` returns `-INF`, and `-INF` returns `INF`.

Unary plus (+x)

The unary plus operator returns its operand with the sign unchanged. It makes no change to the operand, but it is included for completeness. Changing the stylesheet we used for unary minus, we get these results:

```
An example of the unary plus operator:
    $x = -10
   +$x = -10
```

Boolean Operators

XPath provides several boolean operators. They're all straightforward (in XPath 1.0, anyway), so we'll just list them here.

When you're working with boolean expressions in XPath, remember that the values `'true'` and `'false'` are just strings. If you need to use the boolean values, use the functions `true( )` and `false( )`. Simply using `true` in an XPath expression means a node whose name is `true`, which is almost certainly not what you want. To emphasize the point, we'll refer to the boolean values with their functions.

As we saw in the section on mathematical operators, converting `true( )` and `false( )` to numbers returns `1` and `0`, respectively. XPath defines rules for converting different datatypes to boolean values. See "Converting to boolean values" in Chapter 5 for all the details.

Comparing expressions

There are several operators from XPath 1.0 (and 2.0) that compare expressions. We'll look at those here:

`=` *(equal)*
: Given two expressions, returns `true( )` if the two expressions evaluate to the same value, and returns `false( )` otherwise.

`!=` *(not equal)*
: Given two expressions, returns `true( )` if the two expressions do not evaluate to the same value, and returns `false( )` otherwise.

`<` *or* `<` *(less than)*
: Given two expressions, returns `true( )` if the first expression evaluates to a value less than the second. Otherwise, it returns `false( )`. The less than operator is usually escaped with `<` so that it doesn't look like the opening arrow of an XML tag.

`<=` *or* `<=` *(less than or equal)*
: Given two expressions, returns `true( )` if the first expression evaluates to a value less than or equal to the second. Otherwise, it returns `false( )`. The less than or equal operator is usually escaped with `<=` so that it doesn't look like the start of an XML element named `=`.

`>` *or* `>` *(greater than)*
: Given two expressions, returns `true( )` if the first expression evaluates to a value greater than the second. Otherwise, it returns `false( )`. The greater than operator can be escaped with `>` so that it doesn't look like the closing arrow of an XML tag.

`>=` *or* `>=` *(greater than or equal)*
: Given two expressions, returns `true( )` if the first expression evaluates to a value greater than or equal to the second. Otherwise, it returns `false( )`. The greater than or equal operator can be escaped with `>=`.

`and`
: Given two expressions, returns `true( )` if both expressions evaluate to `true( )`. If either evaluates to `false( )`, then `and` returns `false( )`.

`or`
: Given two expressions, returns `true( )` if either expression evaluates to `true( )`. If both values evaluate to `false( )`, then `or` returns `false( )`.

For a boolean `not` operation, XPath provides the `not( )` function. The `not` keyword doesn't exist in XPath. See the `not( )` function in Appendix C for the details.

Many programming languages define specific rules for evaluating boolean operators such as `and` and `or`. For example, many languages state that the first term is evaluated before the second, and that the second term is not evaluated if the overall result is known after the first term is evaluated. (If the first term evaluates to `false( )`, the result of an `and` expression is `false( )`; if the first term evaluates to `true( )`, the result of an `or` expression is `true( )`.) XPath does not define these rules, so XSLT processors are free to implement `and` and `or` as they see fit.

[2.0] Comparing atomic values

As we discussed earlier, XPath 2.0 introduces the concept of atomic values. There are six new operators that allow us to compare values:

`eq`
: Determines whether two values are equal.

`ge`
: Determines whether the first value is greater than or equal to the second.

`gt`
: Determines whether the first value is greater than the second.

`le`
: Determines whether the first value is less or equal to the second.

`lt`
: Determines whether the first value is less than the second.

`ne`
: Determines whether two values are not equal.

The `eq` and `ne` operators work on any of the following datatypes:

- Numeric values (`xs:decimal`, `xs:double`, `xs:float`, `xs:integer`)

- Boolean values
- Durations (`xs:duration`, `xs:yearMonthDuration`, `xs:dayTimeDuration`)
- Dates and times (`xs:date`, `xs:time`, `xs:dateTime`)
- Parts of dates (`xs:gYear`, `xs:gYearMonth`, `xs:gMonth`, `xs:gMonthDay`, `xs:gDay`)
- QNames (`xs:QName`)
- Binary data (`xs:hexBinary`, `xs:base64Binary`)
- Notations (`xs:NOTATION`)

The `ge`, `gt`, `le`, and `lt` operators support fewer datatypes:

- Numeric values (`xs:decimal`, `xs:double`, `xs:float`, `xs:integer`)
- Boolean values
- Durations (`xs:duration`, `xs:yearMonthDuration`, `xs:dayTimeDuration`)
- Dates and times (`xs:date`, `xs:time`, `xs:dateTime`)

The operators for comparing values work slightly differently than the general comparisons. Value comparisons are stricter because they require the two operands to be the same type. Given these XML elements:

```
...
<brand>
  <name>Callebaut</name>
  <units>8203</units>
</brand>
...
```

the expression `brand[1]/units gt 10000` raises an error because we're comparing an untyped value and an integer. We need to compare two integers here, so we must either convert the value of the `<units>` element to a number or use an XML Schema to identify the element as an `xs:integer`. If we use the general comparison operators to compare the values (`brand[1]/units > 10000`), it works just fine. To use the value comparison operators, the expression `xs:integer(brand[1]) gt 10000` works as well, although it requires us to cast the value ourselves.

The general comparison operators (`=`, `!=`, `<=`, `>=`, `<` and `>`) can compare nodes and sequences in addition to values, so you'll probably use them far more often. Just be aware that XPath 2.0 gives you new comparison operators that work on atomic values; they're very useful if you're working with values instead of nodes or sequences.

[2.0] Comparing sequences

And speaking of comparing sequences, that works differently than you might think. When comparing a sequence to a value, the XSLT processor compares the value to each value in the sequence. If the comparison is true for *any* value in the sequence, the operation returns *true*. We'll use an XML document of chocolate sales for our examples:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

Here's the stylesheet we'll use to demonstrate comparisons with sequences:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- compare-sequences.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Comparing sequences with values:</xsl:text>
    <xsl:text>&#xA;  Sales figures (/report/brand/units):</xsl:text>s
    <xsl:text>&#xA;    </xsl:text>
    <xsl:value-of select="/report/brand/units" separator=", "/>
    <xsl:text>&#xA;&#xA;      /report/brand/units &gt;  27408 : </xsl:text>
    <xsl:value-of select="/report/brand/units &gt; 27408"/>
    <xsl:text>&#xA;      /report/brand/units &gt;= 27408 : </xsl:text>
    <xsl:value-of select="/report/brand/units &gt;= 27408"/>
    <xsl:text>&#xA;      /report/brand/units &lt;  8203  : </xsl:text>
    <xsl:value-of select="/report/brand/units &lt; 8203"/>
    <xsl:text>&#xA;      /report/brand/units &lt;= 8203  : </xsl:text>
    <xsl:value-of select="/report/brand/units &lt;= 8203"/>
    <xsl:text>&#xA;      /report/brand/units =  22101 : </xsl:text>
    <xsl:value-of select="/report/brand/units = 22101"/>
    <xsl:text>&#xA;      /report/brand/units =  17905 : </xsl:text>
    <xsl:value-of select="/report/brand/units = 17905"/>
    <xsl:text>&#xA;  not(/report/brand/units =  17905): </xsl:text>
    <xsl:value-of select="not(/report/brand/units = 17905)"/>
```

```
    <xsl:text>&#xA;&#xA;Comparing two sequences:</xsl:text>
    <xsl:variable name="testSequence1" as="xs:integer*"
      select="(8203, 22101, 27408, 19268, 14336)"/>
    <xsl:text>&#xA;  $testSequence1 (xs:integer*):&#xA;    </xsl:text>
    <xsl:value-of select="$testSequence1" separator=", "/>
    <xsl:text>&#xA;    $testSequence1 = /report/brand/units: </xsl:text>
    <xsl:value-of select="$testSequence1 = /report/brand/units"/>

    <xsl:variable name="testSequence2" as="xs:integer*"
      select="(19268, 17, 95, 6, 42)"/>
    <xsl:text>&#xA;&#xA;  $testSequence2 (xs:integer*):&#xA;    </xsl:text>
    <xsl:value-of select="$testSequence2" separator=", "/>
    <xsl:text>&#xA;    $testSequence2 = /report/brand/units: </xsl:text>
    <xsl:value-of select="$testSequence2 = /report/brand/units"/>
    <xsl:text>&#xA;    $testSequence2 &lt; /report/brand/units: </xsl:text>
    <xsl:value-of select="$testSequence2 &lt; /report/brand/units"/>
    <xsl:text>&#xA;    $testSequence2 &gt; /report/brand/units: </xsl:text>
    <xsl:value-of select="$testSequence2 &gt; /report/brand/units"/>

    <xsl:variable name="testSequence3" as="xs:string*"
      select="('blue', 'white', '19268')"/>
    <xsl:text>&#xA;&#xA;  $testSequence3 (xs:string*):&#xA;    </xsl:text>
    <xsl:value-of select="$testSequence3" separator=", "/>
    <xsl:text>&#xA;    $testSequence3 = /report/brand/units: </xsl:text>
    <xsl:value-of select="$testSequence3 = /report/brand/units"/>

    <xsl:variable name="testSequence4" as="xs:yearMonthDuration*"
      select="(xs:yearMonthDuration('P3Y8M'),
               xs:yearMonthDuration('P4Y8M'),
               xs:yearMonthDuration('P2Y9M'))"/>
    <xsl:text>&#xA;&#xA;  $testSequence4 (xs:yearMonthDuration*):</xsl:text>
    <xsl:text>&#xA;   (</xsl:text>
    <xsl:value-of select="$testSequence4" separator=", "/>
    <xsl:text>)&#xA;  $testSequence4 &gt; </xsl:text>
    <xsl:text>xs:yearMonthDuration('P4Y7M'): </xsl:text>
    <xsl:value-of select="$testSequence4 > xs:yearMonthDuration('P4Y7M')"/>
  </xsl:template>

</xsl:stylesheet>
```

The results look like this:

```
Comparing sequences with values:
  Sales figures (/report/brand/units):
    27408, 8203, 22101, 14336, 19268

      /report/brand/units >  27408 : false
      /report/brand/units >= 27408 : true
      /report/brand/units <  8203  : false
      /report/brand/units <= 8203  : true
      /report/brand/units =  22101 : true
      /report/brand/units =  17905 : false
  not(/report/brand/units =  17905): true

Comparing two sequences:
  $testSequence1 (xs:integer*):
```

```
  (8203, 22101, 27408, 19268, 14336)
$testSequence1 = /report/brand/units: true

$testSequence2 (xs:integer*):
  (19268, 17, 95, 6, 42)
$testSequence2 = /report/brand/units: true
$testSequence2 < /report/brand/units: true
$testSequence2 > /report/brand/units: true

$testSequence3 (xs:string*):
  (blue, white, 19268)
$testSequence3 = /report/brand/units: true

$testSequence4 (xs:yearMonthDuration*):
 (P3Y8M, P4Y8M, P2Y9M)
$testSequence4 > xs:yearMonthDuration('P4Y7M'): true
```

In the first set of comparisons, we're comparing the sequence matched by `/report/brand/units` to individual values. The comparison `/report/brand/units > 27408` is `false` because there are no values in the sequence greater than `27408`. When we change the operator to `>=`, the comparison is `true`. For test of equality, if any value in the sequence matches the value we're comparing, the result is `true`. Also notice that we used the `not( )` function to reverse the result here; the `!=` operator would do the same thing.

For the next three examples, we're comparing two sequences. In the first, we're comparing the sequence `$testSequence1` to the sequence matched by `/report/brand/units`. The two sequences have the same five values, although they're in different orders. The two sequences are considered equal in this comparison, although only one value has to match for that to be true. Comparing the sequence from the XML document to `$testSequence2` makes this clear; although the two sequences have only one value in common, they are considered equal. In addition, comparing `$testSequence2` with other operators, we can see that `$testSequence` is less than, equal to, and greater than the sequence matched by `/report/brand/units`. This isn't intuitive, but that's how it works.

The difference in the third example is that `$testSequence3` is a sequence of `xs:string`s. Because we're not using an XML Schema, the values from the XML document are untyped. The XSLT processor automatically converted the values from the document to `xs:integer`s in the previous two examples, and it converts them to `xs:string`s here. The sequence `('blue', 'white', '19268')` matches because converting the untyped value of the parsed `<units>19268</units>` element to a string creates the value `'19268'`.

The final example here demonstrates comparisons with a sequence of `xs:yearMonthDuration`s. We create a sequence of three values, and then compare it to the value `xs:yearMonthDuration('P4Y7M')`. Because one of the values is greater than 4 years, 7 months, this returns `true`.

More sophisticated operations on sequences are possible. See the sections "[2.0] Quantified Expressions—some and every" and "[2.0] Set Operators—except, intersect, and union" later in this chapter.

[2.0] Conditional Expressions—if, then, and else

One of the less elegant features of XSLT is its if-then-else logic. If I want to test one condition (a simple `if`), I use `<xsl:if>`. If I want to change that to test more than one condition or add an `else` case, I have to use `<xsl:choose>`, `<xsl:when>`, and `<xsl:otherwise>`. (We cover those elements in Chapter 5.) XPath 2.0 gives us the extremely useful `if` operator. We can now do if-then-else logic inside the XPath expression itself.

For comparison, here's how we do things in XSLT 1.0:

```
<?xml version="1.0"?>
<!-- if-1_0.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:param name="x" select="'10'"/>

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;An example of if-then-else logic in XSLT 1.0:</xsl:text>
    <xsl:text>&#xA;&#xA;  If $x is larger than 10, print 'Big', </xsl:text>
    <xsl:text>&#xA;    otherwise print 'Little'</xsl:text>
    <xsl:text>&#xA;&#xA;           </xsl:text>
    <xsl:choose>
      <xsl:when test="$x &gt; 10">
        <xsl:text>Big</xsl:text>
      </xsl:when>
      <xsl:otherwise>
        <xsl:text>Little</xsl:text>
      </xsl:otherwise>
    </xsl:choose>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

We look at the value of `$x` and write `Big` if it's larger than 10; otherwise, we write `Little`. Pretty simple stuff, but the `<xsl:choose>` element takes up 8 lines here. To do the same thing in XSLT 2.0, it's much simpler:

```
<?xml version="1.0"?>
<!-- if-2_0.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:param name="x" select="10"/>

  <xsl:output method="text"/>

  <xsl:template match="/">
    ...
    <xsl:value-of select="if ($x &gt; 10) then 'Big' else 'Little'"/>
```

```
    ...
  </xsl:template>

</xsl:stylesheet>
```

We've accomplished the same result with a single XPath expression.

There are two important details to remember when using the `if` operator: the expression we're testing must be enclosed in parentheses, and we *always* have to use the `else` keyword.

[2.0] Iterators Over Sequences—The for Operator

Given XSLT 2.0's emphasis on sequences, it makes sense that we would have an operator to iterate through all the values of a sequence. Just as the `if` operator lets you put the logic of `<xsl:choose>` into an XPath expression, the `for` operator gives XPath expressions the power of `<xsl:for-each>`. Here is a stylesheet with an example:

```
<?xml version="1.0"?>
<!-- for.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:variable name="English-months" as="xs:string*"
    select="('January', 'February', 'March', 'April',
             'May', 'June', 'July', 'August',
             'September', 'October', 'November',
             'December')"/>
  <xsl:variable name="German-months" as="xs:string*"
    select="('Januar', 'Februar', 'März', 'April',
             'Mai', 'Juni', 'Juli', 'August',
             'September', 'Oktober', 'November',
             'Dezember')"/>

  <xsl:template match="/">
    <xsl:value-of
      select="for $m in ($English-months, $German-months) return
                if (starts-with($m, 'J'))
                  then concat ($m, ' starts with J!&#xA;')
                  else ''"
      separator=""/>
  </xsl:template>

</xsl:stylesheet>
```

[2.0] Quantified Expressions—some and every

XPath 2.0 provides the `some` and `every` operators to perform a test against a sequence. The `some` operator returns `true` if the test is true for at least one item in the sequence,

while every returns false if the test is false for at least one item in the sequence. Here is an example of the two operators:

```
<?xml version="1.0"?>
<!-- some-every.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:variable name="English-months" as="xs:string*"
    select="('January', 'February', 'March', 'April',
             'May', 'June', 'July', 'August',
             'September', 'October', 'November',
             'December')"/>

  <xsl:template match="/">
    <xsl:text>&#xA;An example of the XPath 2.0 every and </xsl:text>
    <xsl:text>some operators:&#xA;&#xA;</xsl:text>
    <xsl:text>  If ANY month name has a string-length() </xsl:text>
    <xsl:text>&#xA;    greater than 4, print 'Yes,' otherwise</xsl:text>
    <xsl:text>&#xA;    print 'No'&#xA;&#xA;         </xsl:text>

    <xsl:value-of
      select="if (some $m in $English-months satisfies
              (string-length($m) &gt; 4)) then 'Yes' else 'No'"/>

    <xsl:text>&#xA;</xsl:text>
    <xsl:text>&#xA;&#xA;  If EVERY month name has a string-</xsl:text>
    <xsl:text>length() &#xA;    greater than 4, print 'Yes,' </xsl:text>
    <xsl:text>otherwise&#xA;    print 'No'&#xA;&#xA;         </xsl:text>

    <xsl:value-of
      select="if (every $m in $English-months satisfies
              (string-length($m) &gt; 4)) then 'Yes' else 'No'"/>

    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

We have a sequence containing the months of the year in English, and we use some and every against that sequence. We're testing to see whether each month name has a string-length( ) greater than 4. The some expression returns true, while the every expression returns false. This stylesheet is written to illustrate the new operands; it doesn't use anything from an XML document. In a more normal case, we would use some and every against an XPath expression that selected nodes from an XML document. If you replace the variable $English-months with /sonnet/lines/line in the two preceding expressions, both will evaluate to true, assuming you use the stylesheet to process sonnet.xml.

Complications of the some and every Operators

It's entirely possible that the sequence you're testing is empty. If your expression is `some $m in /sonnet/lines/words` (or anything that returns an empty sequence), the `some` operator returns `false`, while the `every` operator returns `true`. This less-than-intuitive result is defined by the XPath 2.0 and XQuery 1.0 Functions and Operators specification (Section 3.9, if you want to take a look). These results for empty sequences make more sense if you think about a simple way to evaluate these operators:

- For the `some` operator, assume the result is `false`. As we evaluate the test against each item in the sequence, if the test is ever `true`, we stop evaluating the items and return `true`. If we get to the end of the sequence and we haven't found any item for which the test is `true`, we'll return `false`.
- For the `every` operator, assume the result is `true`. As we evaluate the test against each item in the sequence, if the test is ever `false`, we stop evaluating the items and return `false`. If we get to the end of the sequence and we haven't found any item for which the test is `false`, we'll return `true`.

A complication of this rule is that a `some` or `every` expression might contain a fatal error if all of the items in the test sequence were evaluated. The XPath spec does not define any required behavior here. If an XSLT processor tests items only until it knows the value of `some` or `every`, those errors might never be encountered. Here are two examples from the XPath spec:

```
some $x in (1, 2, "cat") satisfies $x * 2 = 4
every $x in (1, 2, "cat") satisfies $x * 2 = 4
```

If an XSLT processor evaluates the items in the sequence from the first to the last, and it stops evaluating items as soon as it determines the final result, the fatal error `"cat" * 2` won't happen. The test is `true` for the value `2`, so `some` returns `true`. The test is `false` for the value `1`, so `every` returns `false`.

The XPath specification does not say how a processor must implement `some` and `every`, nor does it specify the order in which items in the sequence are evaluated. A processor's internal structures might make it simpler to start at the end of the sequence; if so, both of these expressions would generate a fatal error. The most ominous implication of this behavior is that the fatal error might happen intermittently based on the input data. You can avoid this by using the datatyping operators, which we'll discuss soon. (You can also avoid this if your data has been validated before it gets to the XSLT processor, assuming you trust your data source implicitly....)

[2.0] Range Expressions—The to Operator

To create sequences of integers, XPath 2.0 introduces the `to` operator. For example, here is a short stylesheet that creates a sequence of five integers and a reversed sequence of five integers, then prints the values of the sequences:

```
<?xml version="1.0"?>
<!-- to.xsl -->
```

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:variable name="some-numbers" as="xs:integer*"
    select="1 to 5"/>

  <xsl:variable name="reversed-numbers" as="xs:integer*"
    select="reverse(1 to 5)"/>

  <xsl:template match="/">
    <xsl:value-of select="$some-numbers" separator=", "/>
    <xsl:text>
</xsl:text>
    <xsl:value-of select="$reversed-numbers" separator=", "/>
  </xsl:template>

</xsl:stylesheet>
```

The `to` operator creates only sequences of integers. If the second number is lower than the first (`10 to 1`, for example), the result is an empty sequence. If the second number is the same as the first (such as `10 to 10`), the result is a sequence that contains that single integer.

Be aware that the `to` operator can be used as part of a larger sequence. For example, we could create the sequence `(1 to 17, 65 to 100)`. Finally, if you need to create a sequence of numbers in descending order, you can use the [2.0] `reverse( )` function (see Appendix C) to reverse the sequence of integers created with the `to` operator.

The results of this stylesheet look like this:

```
1, 2, 3, 4, 5
5, 4, 3, 2, 1
```

[2.0] Constructor Functions

XPath 2.0 introduces the idea of *constructor functions*, which are similar to constructor functions in object-oriented languages. As with object-oriented languages, the name of the constructor function is the name of the datatype. Here's how to create a value of type `xs:date`:

```
<xsl:variable name="birthday" select="xs:date('1995-04-21')"/>
```

This takes the string `1995-04-21` and creates a new value of type `xs:date`. This can cause runtime errors, as you would expect. A stylesheet that contains this instruction won't run at all:

```
<xsl:variable name="birthday" select="xs:date('next Tuesday')"/>
```

This is a static error because the stylesheet processor knows this instruction will fail. On the other hand, we'll get a runtime error if we send bad data to the `xs:date` constructor while the stylesheet is being processed:

```
<xsl:variable name="birthday" select="xs:date(@birthday)"/>
```

This uses the `birthday` attribute of the current node to create a new `xs:date` value. If that attribute contains a value that can be cast as a `xs:date`, everything is fine; if the attribute doesn't contain valid data, we get a runtime error.

[2.0] Datatype Operators—instance of, castable as, cast as, and treat as

As you would expect, a language that supports datatypes and constructors also has operators to convert a value from one type to another. XPath 2.0 provides four operators: `instance of`, `castable as`, `cast as`, and `treat as`. We'll cover those here.

instance of

The `instance of` operator lets us see whether a value is an instance of a particular datatype. Here are some examples:

```
<?xml version="1.0"?>
<!-- instance-of.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Some tests of the "instance of" operator:</xsl:text>

    <xsl:text>&#xA;&#xA;  '1995-04-21' instance of xs:date: </xsl:text>
    <xsl:value-of select="'1995-04-21' instance of xs:date"/>
    <xsl:text>&#xA;  xs:date('1995-04-21') instance of xs:date: </xsl:text>
    <xsl:value-of select="xs:date('1995-04-21') instance of xs:date"/>
    <xsl:text>&#xA;&#xA;  3 instance of xs:integer: </xsl:text>
    <xsl:value-of select="3 instance of xs:integer"/>
    <xsl:text>&#xA;  '3' instance of xs:integer: </xsl:text>
    <xsl:value-of select="'3' instance of xs:integer"/>
    <xsl:text>&#xA;  number('3') instance of xs:integer: </xsl:text>
    <xsl:value-of select="number('3') instance of xs:integer"/>
    <xsl:text>&#xA;  number('3') instance of xs:double: </xsl:text>
    <xsl:value-of select="number('3') instance of xs:double"/>
    <xsl:text>&#xA;  xs:integer('3') instance of xs:integer: </xsl:text>
    <xsl:value-of select="xs:integer('3') instance of xs:integer"/>
    <xsl:text>&#xA;  'e' instance of xs:integer: </xsl:text>
    <xsl:value-of select="'e' instance of xs:integer"/>

  </xsl:template>
</xsl:stylesheet>
```

The results of the stylesheet are:

```
Some tests of the "instance of" operator:

  '1995-04-21' instance of xs:date: false
  xs:date('1995-04-21') instance of xs:date: true

  3 instance of xs:integer: true
  '3' instance of xs:integer: false
  number('3') instance of xs:integer: false
  number('3') instance of xs:double: true
  xs:integer('3') instance of xs:integer: true
  'e' instance of xs:integer: false
```

For the first test, we're asking whether the string value `1995-04-21` is an instance of `xs:date`. This is false; XPath doesn't automatically try to cast the string as an `xs:date`. In the second test, we use the `xs:date` constructor function to create a new `xs:date` value with the string. This test is true.

The last six tests here check the `xs:integer` datatype. The atomic value `3`, as we would expect, is an `xs:integer`. The string `'3'` is not an `xs:integer`, even though we can cast it as one (we'll look at casting next). In addition to casting, another way to convert a string to a numeric value is to use the `number( )` function. The `number( )` function returns an `xs:double`, so `instance of xs:integer` is false. Using the constructor function `xs:integer( )` returns an `xs:integer`, of course. Finally, the value `'e'` isn't an `xs:integer` and can't be cast as one.

cast as

The `instance of` operator tells us about the datatype of a value; it doesn't actually convert the value to the appropriate datatype. There are times when we want to take a value of a particular datatype and create the equivalent value of another. In other programming languages, this is done by *casting*. XPath 2.0 provides the `cast as` operator to do just that. Here are a few examples:

```
<?xml version="1.0"?>
<!-- cast-as.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Some tests of the "cast as" operator:</xsl:text>

    <xsl:text>&#xA;&#xA;  '1995-04-21' cast as xs:date: </xsl:text>
    <xsl:value-of select="'1995-04-21' cast as xs:date"/>
    <xsl:text>&#xA;  '3' cast as xs:integer: </xsl:text>
    <xsl:value-of select="'3' cast as xs:integer"/>
    <xsl:text>&#xA;  3 cast as xs:integer: </xsl:text>
    <xsl:value-of select="3 cast as xs:integer"/>
```

```
    <xsl:text>&#xA;  'e' cast as xs:integer: </xsl:text>
    <xsl:text>[causes a fatal error if we try it]</xsl:text>

  </xsl:template>
</xsl:stylesheet>
```

In this sample, we cast the string `1995-04-21` to an `xs:date`. This works without a hitch, as do the next two tests. The fourth test, `'3' cast as xs:integer`, fails if we try to execute it. The stylesheet won't run at all because this is a static error. Here are the results:

```
Some tests of the "cast as" operator:

  '1995-04-21' cast as xs:date: 1995-04-21
  '3' cast as xs:integer: 3
  3 cast as xs:integer: 3
  'e' cast as xs:integer: [causes a fatal error if we try it]
```

As you'd expect, it's a dynamic error if we extract a value from an XML source document and try to cast it to a datatype that won't work. If only we had a way to see if `cast as` would work before we actually tried it, we could avoid dynamic errors....

castable as

The `castable as` operator lets us see whether a given cast will work. If it won't, we can respond to that gracefully instead of having our stylesheet fail. Here's how we use `castable as`:

```
<?xml version="1.0"?>
<!-- castable-as.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Some tests of the "castable as" operator:</xsl:text>

    <xsl:text>&#xA;&#xA;  '1995-04-21' castable as xs:date: </xsl:text>
    <xsl:value-of select="'1995-04-21' castable as xs:date"/>
    <xsl:text>&#xA;  '3' castable as xs:integer: </xsl:text>
    <xsl:value-of select="'3' castable as xs:integer"/>
    <xsl:text>&#xA;  3 castable as xs:integer: </xsl:text>
    <xsl:value-of select="3 castable as xs:integer"/>
    <xsl:text>&#xA;  'e' castable as xs:integer: </xsl:text>
    <xsl:value-of select="'e' castable as xs:integer"/>

  </xsl:template>
</xsl:stylesheet>
```

With the `castable as` operator, we can see whether a cast will work before we try it. If it's not going to work (we can't convert `e` to an `xs:integer`), we can do something else in our stylesheet. Here are the results from our stylesheet:

```
Some tests of the "castable as" operator:

  '1995-04-21' castable as xs:date: true
  '3' castable as xs:integer: true
  3 castable as xs:integer: true
  'e' castable as xs:integer: false
```

treat as

There is one more operator XPath gives us: the `treat as` operator. This one is a little harder to grasp. To understand why it's needed, we'll have to talk about *static types* and *dynamic types*. A static datatype is the type of a value as it is declared. A dynamic type may be more specific than the static type; more often they're the same. To take an example from the XPath 2.0 spec, the static type of a variable could be `xs:integer*`, yet its dynamic type could be `xs:integer`. Here's an example:

```
<xsl:variable name="integerSequence" as="xs:integer+" select="(3)"/>
```

The variable `$integerSequence` is a sequence of one or more integers. In this instance, however, `$integerSequence` is also a single integer. Here's an example in which we treat a sequence of integers as a single integer:

```
<?xml version="1.0"?>
<!-- treat-as.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:variable name="numbers" as="xs:integer*">
    <xsl:sequence select="/numbers/number"/>
  </xsl:variable>

  <xsl:template match="/">
    <xsl:variable name="number" as="xs:integer"
      select="$numbers treat as xs:integer"/>

    <xsl:text>&#xA;An example of the XPath 2.0 treat as </xsl:text>
    <xsl:text>operator:&#xA;&#xA;</xsl:text>
    <xsl:text>  Treat a sequence of integers as a single integer:</xsl:text>
    <xsl:text>&#xA;         </xsl:text>

    <xsl:value-of select="$number"/>
  </xsl:template>

</xsl:stylesheet>
```

Here we have a variable that consists of a sequence of values of `<number>` elements in an XML document. If there is only one number in the sequence, the stylesheet works. If the sequence isn't a singleton (it's empty or has more than one integer), we get a runtime error. This XML document, for example, doesn't cause any problems:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- numbers.xml -->
<numbers>
  <number>3</number>
</numbers>
```

Adding more `<number>` elements (or using a document that doesn't have any `<number>` elements at all) causes a runtime error. The `treat as` operator tells the XSLT processor not to worry about the inconsistencies between datatypes (`xs:integer*` versus `xs:integer`), but it puts the burden on us to make sure the dynamic type of the data is correct. To do that, of course, we can use the `cast as` and `castable as` operators.

[2.0] Set Operators—except, intersect, and union

One weakness of XPath 1.0 was the inability to compare sets of nodes. If we selected two node-sets with two XPath 1.0 expressions, it was difficult to tell which nodes were in both sets and which were in one set but not the other. XPath 2.0 has two new set operators, `except` and `intersect`. If you enjoy extra typing, XPath 2.0 also adds the `union` operator, a synonym for the vertical bar operator (|) in XPath 1.0. (The vertical bar is still supported in XPath 2.0.)

Be aware that these operators work only on sequences of nodes. If you try to use them with sequences that contain atomic values, you'll get an error.

As we discuss these operators, we'll use this XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- books.xml -->
<favorite-books>
  <booklist>
    <book isbn="0596000537"
      favorite="Doug Sheri">XSLT</book>
    <book isbn="0141439777"
      favorite="Doug">Tristram Shandy</book>
    <book isbn="0142437298"
      favorite="Doug">Herzog</book>
    <book isbn="0679762108"
      favorite="Doug Sheri">The Sportswriter</book>
    <book isbn="0143035479"
      favorite="Sheri">The Girls' Guide to Hunting and Fishing</book>
    <book isbn="0375724443"
      favorite="Sheri">Ava's Man</book>
  </booklist>
</favorite-books>
```

We'll select two sequences in our sample stylesheets: the books I like (the `favorite` attribute contains the string `Doug`) and the books my wife likes (the `favorite` attribute contains the string `Sheri`). We'll use those two sequences to illustrate the set operators.

The `except`, `intersect`, and `union` operators compare *the nodes themselves*, not the values of those nodes. If two different nodes have the same value, they are still two different nodes. These operators help you select different nodes, not different values. If you're curious about the values of the nodes, the new [2.0] `distinct-values( )` function (see Appendix C) will probably be useful.

Here is a sample stylesheet for the `except` operator:

```
<?xml version="1.0"?>
<!-- except.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:variable name="Dougs-favorites" as="node()*">
    <xsl:sequence
      select="/favorite-books/booklist
              /book[contains(@favorite, 'Doug')]"/>
  </xsl:variable>

  <xsl:variable name="Sheris-favorites" as="node()*">
    <xsl:sequence
      select="/favorite-books/booklist
              /book[contains(@favorite, 'Sheri')]"/>
  </xsl:variable>

  <xsl:template match="/">
    <xsl:text>&#xA;Books Doug likes but Sheri doesn't:</xsl:text>
    <xsl:text>&#xA;&#xA;  </xsl:text>

    <xsl:for-each select="$Dougs-favorites except $Sheris-favorites">
      <xsl:sort select="."/>
      <xsl:value-of select="."/>
      <xsl:text>&#xA;  </xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

The other sample stylesheets are identical, but they use the `intersect` and `union` operators instead.

except

The `except` operator returns all of the nodes that are in the first sequence but not in the second. Given our two sequences, the expression `$Dougs-favorites except $Sheris-favorites` generates these results:

```
Books Doug likes but Sheri doesn't:
```

```
  Herzog
  Tristram Shandy
```

intersect

Given two sequences of nodes, `intersect` returns a sequence containing all of the nodes in both sequences. *All duplicate nodes are removed*; each node appears only once. The `intersect` operator returns these results:

```
<!-- intersect.xsl -->
...
    <xsl:for-each select="$Dougs-favorites intersect $Sheris-favorites">
...

Books we both like:

  The Sportswriter
  XSLT
```

union

The `union` operator returns a node-set containing all nodes in both sets. As with the `intersect` operator, all duplicates are removed. Using the `union` operator gives us these results:

```
<!-- union.xsl -->
...
    <xsl:for-each select="$Dougs-favorites union $Sheris-favorites">
...

All the books we like:

  Ava's Man
  Herzog
  The Girls' Guide to Hunting and Fishing
  The Sportswriter
  Tristram Shandy
  XSLT
```

As we noted earlier, the vertical bar (|) operator is still supported. Changing the stylesheet from:

```
<xsl:for-each select="$Dougs-favorites union $Sheris-favorites">
```

to:

```
<xsl:for-each select="$Dougs-favorites | $Sheris-favorites">
```

generates the same results.

To emphasize the point that these operators compare nodes, not their values, we could change the structure of our list of books:

```
<favorite-books>
  <favorites person="Doug">
    <book isbn="0679762108">The Sportswriter</book>
```

```
  </favorites>
  <favorites person="Sheri">
    <book isbn="0679762108">The Sportswriter</book>
  </favorites>
</favorite-books>
```

The two `<book>` elements here have identical values, but they are two different nodes. The `except`, `intersect`, and `union` operators treat them as such.

[2.0] Node Operators

XPath 2.0 defines three new operators to work with nodes: the `is` operator, the node-before operator (`<<`), and the node-after operator (`>>`).

The is operator

The `is` operator compares two nodes to see whether they are the same. This compares the nodes themselves, not their values. Continuing our example from the previous section, here's a stylesheet that illustrates how the operator works:

```
<?xml version="1.0"?>
<!-- is.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:variable name="Dougs-favorites" as="node()*">
    <xsl:sequence
      select="/favorite-books/booklist
              /book[contains(@favorite, 'Doug')]"/>
  </xsl:variable>

  <xsl:variable name="Sheris-favorites" as="node()*">
    <xsl:sequence
      select="/favorite-books/booklist
              /book[contains(@favorite, 'Sheri')]"/>
  </xsl:variable>

  <xsl:template match="/">
    <xsl:text>A test of the is operator:</xsl:text>
    <xsl:text>&#xA;  Comparing the first nodes of </xsl:text>
    <xsl:text>the sequences:&#xA;</xsl:text>

    <xsl:value-of
      select="if (subsequence($Dougs-favorites, 1, 1) is
                  subsequence($Sheris-favorites, 1, 1))
              then '    The first nodes are the same!&#xA;'
              else '    The first nodes aren''t the same!&#xA;'"/>

    <xsl:text>  Reversing one sequence and trying it </xsl:text>
    <xsl:text>again:&#xA;</xsl:text>
```

```
    <xsl:value-of
      select="if (subsequence($Dougs-favorites, 1, 1) is
                  subsequence(reverse($Sheris-favorites), 1, 1))
              then '    The first nodes are the same!&#xA;'
              else '    The first nodes aren''t the same!&#xA;'"/>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet creates two sequences of nodes, then compares the first nodes of each sequence. In the first example, the nodes are the same. In the second example, we reverse the order of one of the sequences, so the nodes are not the same. (Of course, if each sequence contained only one item, `is` would still return `true`.) Here are the results:

```
A test of the is operator:

  Comparing the first nodes of the sequences:
    The first nodes are the same!
  Reversing one sequence and trying it again:
    The first nodes aren't the same!
```

node-after (>>)

The node-after operator compares two nodes. If the first node occurs after the second, node-after returns `true`; otherwise, it returns `false`. The expression `node1 >> node1` returns `false`, as you would expect; a node can't appear after itself. Here's a sample stylesheet:

```
<?xml version="1.0"?>
<!-- node-after.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:variable name="Dougs-favorites" as="node()*">
    <xsl:sequence
      select="/favorite-books/booklist
              /book[contains(@favorite, 'Doug')]"/>
  </xsl:variable>

  <xsl:template match="/">
    <xsl:text>A test of the node-after (>>) operator:</xsl:text>
    <xsl:text>&#xA;&#xA;  Comparing nodes from </xsl:text>
    <xsl:text>the sequence:&#xA;</xsl:text>

    <xsl:value-of
      select="if (subsequence($Dougs-favorites, 1, 1) >>
                  subsequence($Dougs-favorites, 2, 1))
              then '    node1 >> node2 = true&#xA;'
              else '    node1 >> node2 = false&#xA;'"/>
    <xsl:value-of
      select="if (subsequence($Dougs-favorites, 2, 1) >>
                  subsequence($Dougs-favorites, 1, 1))
```

```
            then '    node2 >> node1 = true&#xA;'
            else '    node2 >> node1 = false&#xA;'"/>
    <xsl:value-of
      select="if (subsequence($Dougs-favorites, 1, 1) >>
                  subsequence($Dougs-favorites, 1, 1))
            then '    node1 >> node1 = true&#xA;'
            else '    node1 >> node1 = false&#xA;'"/>
  </xsl:template>

</xsl:stylesheet>
```

The nodes in the sequence `$Dougs-favorites` are in document order, so `node1 >> node2` returns `false` and `node2 >> node1` returns `true`. In the third example, we compare the node to itself. A node can't appear before itself, so `node1 >> node1` returns `false`. Here are the results:

```
A test of the node-after (>>) operator:

  Comparing nodes from the sequence:
    node1 >> node2 = false
    node2 >> node1 = true
    node1 >> node1 = false
```

node-before (<<)

The node-before operator compares two nodes and returns `true` if the first node appears in the document before the second. Replacing the node-after operator with the node-before operator in our previous stylesheet gives these results:

```
A test of the node-before (<<) operator:

  Comparing nodes from the sequence:
    node1 << node2 = true
    node2 << node1 = false
    node1 << node1 = false
```

Although the specs define the node-before operator as <<, we have to escape the less-than sign inside an XPath expression. The expressions in the stylesheet use the node-before operator in the form `node1 << node2`.

[2.0] Comments in XPath Expressions

Another addition to the XPath 2.0 syntax is the ability to add comments. Using delightfully happy syntax, a comment begins with `(:` and ends with `:)`. We'll use a stylesheet with a complicated `if` statement:

```
<?xml version="1.0"?>
<!-- comments.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:for-each select="cars/make">
      <xsl:text>&#xA;  Car: </xsl:text>
      <xsl:value-of select="."/>
      <xsl:text> - </xsl:text>
      <xsl:value-of
        select="(: Most of our cars are from North America,
                   so we look there first :)
                if (@geography = 'North America') then
                  'Domestic car'

                (: Next, see if the car is from Europe :)
                else if (@geography = 'Europe') then
                  'Import from Europe'

                (: Check for Asia :)
                else if (@geography = 'Asia') then
                  &quot;It's from Asia&quot;

                (: If it's anything else, just say
                  'We don't know' :)
                else
                  'We don''t know!'"/>

    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet has three `if` statements that check the value of the `geography` attribute of an element. We've used spacing and comments liberally here to make the code more legible. In the last comment, you can see that we don't have to escape quote marks inside the comment, although we do have to handle them appropriately outside the comment. For one quote that contains an apostrophe, we wrap the text in double quotes (`"It's from Asia"`). For the next quote, we use a doubled apostrophe (`'We don''t know!'`) to display the text.

Here's the XML document we'll transform:

```
<?xml version="1.0"?>
<!-- carlist-geography.xml -->
<cars>
  <make geography="Europe">Alfa Romeo</make>
  <make geography="Europe">Bentley</make>
  <make geography="North America">Chevrolet</make>
  <make geography="North America">Dodge</make>
  <make geography="North America">GMC</make>
  <make geography="Asia">Honda</make>
  <make geography="Asia">Isuzu</make>
  <make geography="?">Quantum</make>
</cars>
```

The output looks like this:

```
Car: Alfa Romeo - Import from Europe
Car: Bentley - Import from Europe
Car: Chevrolet - Domestic car
Car: Dodge - Domestic car
Car: GMC - Domestic car
Car: Honda - Import from Asia
Car: Isuzu - Import from Asia
Car: Quantum - We don't know!
```

These comments work only within an XPath statement; you can't use them to comment out parts of your stylesheet. Comments can also be nested. If we wanted to remove the `if` statement that checks for European cars, we could comment out that entire section of the XPath expression:

```
(: (: Next, see if the car is from Europe :)

else if (@geography = 'Europe') then
  'Import from Europe' :)
```

This comment includes our earlier comment and the `if` statement. XPath comments are more convenient than XML comments, which cannot be nested.

[2.0] Types of XSLT 2.0 Processors

The XSLT 2.0 spec defines two types of XSLT 2.0 processors:

Schema-aware processors
: A *schema-aware* XSLT 2.0 processor supports user-defined schemas. In other words, we can use XML Schema to define our own datatypes and document structures, then ask the processor to validate values or nodes against that schema.

Basic processors
: A *basic* XSLT 2.0 processor supports only the datatypes defined in the XML Schema Datatypes spec, with a few extra datatypes added by the XPath 2.0 and XQuery 1.0 Data Model spec. We covered all of those datatypes earlier in this chapter.

For a brief introduction to XML Schema, as well as a short discussion of how schemas are used in XSLT 2.0 stylesheets, see Appendix D.

The XPath View of an XML Document

Before we leave the subject of XPath, we'll look at a stylesheet that generates a pictorial view of a document. The stylesheet has to distinguish between all of the different XPath node types, including any `namespace` nodes.

Output View

Figure 3-1 shows the output of our stylesheet. In this graphical view of the document, the nested HTML tables illustrate which nodes are contained inside of others, as well as the sequence in which these nodes occur in the original document. In the section of the document visible in Figure 3-1, the root of the document contains, in order, two processing instructions and two comments, followed by the `<sonnet>` element and two more comments. The `<sonnet>` element, in turn, contains two attributes and an `<auth:author>` element. The `<auth:author>` element contains a namespace node and an element.

Be aware that if you throw a very large XML document at this stylesheet, you'll get an HTML file with hundreds, perhaps thousands of tables. It's possible that your XSLT processor will run out of memory before it's finished with the document. For example, using this stylesheet to process the XML source of the Function Reference appendix creates an HTML file with more than 17,000 tables.

The Stylesheet

Now we'll take a look at the stylesheet and how it works. The stylesheet creates a number of nested tables to illustrate the XPath view of the document. We begin by writing the basic HTML elements to the output stream, defining some CSS styles and creating a legend for our nested tree view. Having created the legend for our document, we select all the different types of nodes and represent them:

```
<xsl:for-each select="*|comment()|processing-instruction()|text()">
    ...
</xsl:for-each>
```

It's very important to understand the difference between the XPath *document root* and the XML *root element*. In our XML sonnet, there are processing instructions and comments outside the root element. The document root contains those processing instructions and comments in addition to the `<sonnet>` element itself. In a location path, `/` represents the document root, while `/sonnet` represents the root element in our XML document. (The more general expression `/*` represents the root element in any document.)

[2.0] The situation is even more complicated in XPath 2.0, where you have to distinguish between a node type (document node, element node, etc.) and the *role* the node plays: nodes of any type can be at the root of a document. It's only in well-formed XML documents that we can say that the root node is a document node and its only element child is the document element. All that being said, if you're transforming XML documents, this situation will come up very rarely.

The `select` attribute in the template for the document root doesn't include attributes (`@*`) or namespace nodes (`namespace::*`) because those can't be defined on a document

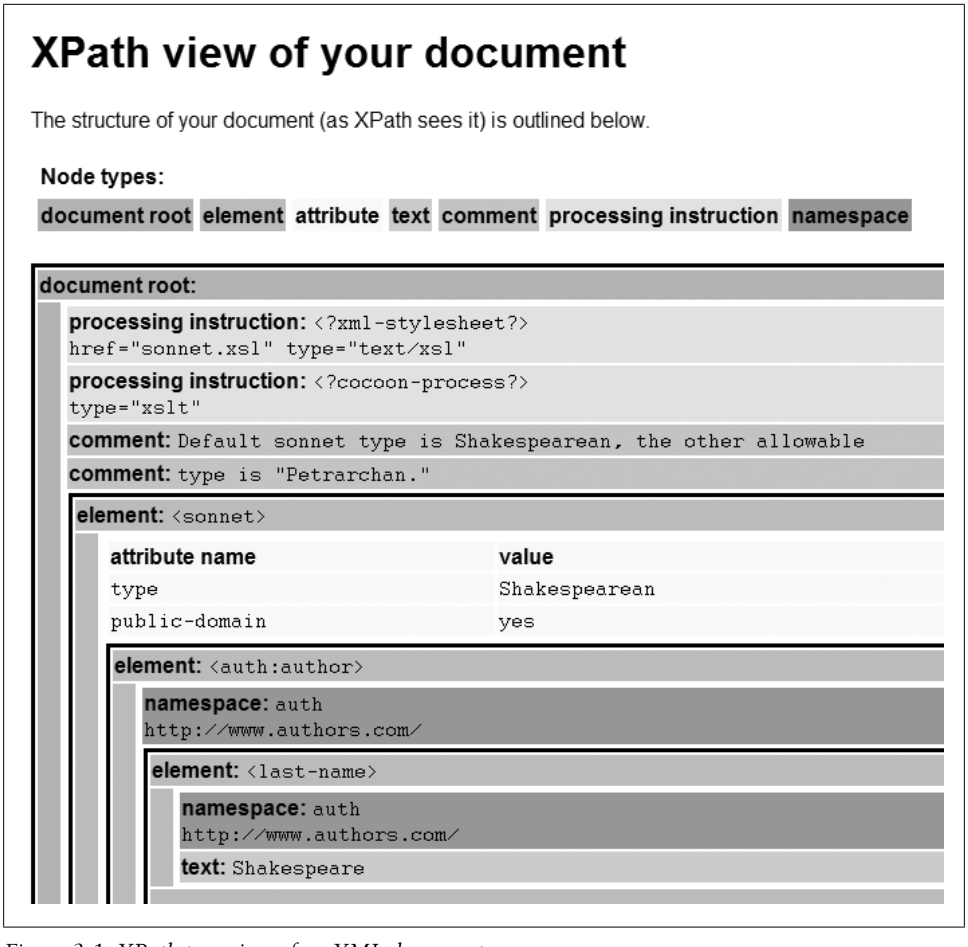


Figure 3-1. XPath tree view of an XML document

root. If you try to select attributes or namespace nodes at the document root, some XSLT processors (Saxon, for example) give you a warning message.

The rest of the stylesheet has templates to handle each of the types of nodes. Most of them print the type of node followed by the content of that node. The only template with any complexity is the element template. We create a new table row, and then put a nested table inside the row. When we create the new table row, we use the HTML `title` and `alt` attributes. Whenever the user pauses his mouse over part of the table, a pop up appears that indicates the ancestry of the element represented in that part of the table. Here's how we create the value for the `title` and `alt` attributes:

```
<xsl:variable name="title">
  <xsl:for-each select="ancestor-or-self::*">
    <xsl:text>/</xsl:text>
    <xsl:value-of select="name()"/>
  </xsl:for-each>
```

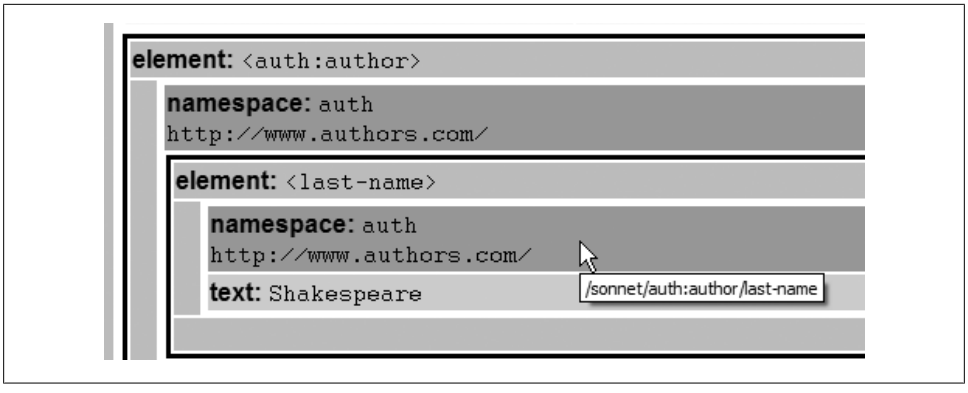


Figure 3-2. Pop-up text shows the ancestry of each element

```
</xsl:variable>
<tr title="{$title}" alt="{$title}">
```

The `ancestor-or-self` axis returns all of the ancestors of the current node plus the current node itself. Even though the `ancestor-or-self` axis returns nodes that occur before the current node, the nodes it returns are in document order. In other words, the first element in the sequence returned by `ancestor-or-self` axis is the root element. Figure 3-2 shows how the pop up looks.

For this example, the sequence of nodes returned by `ancestor-or-self` are the `<sonnet>`, `<auth:author>`, and `<last-name>` elements, in that order.

The next step in processing an element is to look for attributes and namespace definitions. An element is the only type of node that can contain these. Here's how we look for them:

```
<table border="0" width="100%">
  <xsl:if test="count(@*) &gt; 0">
    <tr>
      <td>
        <table width="100%">
          <tr class="attribute">
            <td width="20%">
              <b>attribute name</b>
            </td>
            <td>
              <b>value</b>
            </td>
          </tr>
          <xsl:for-each select="@*">
            <tr class="attribute">
              <td width="20%">
                <span class="literal">
                  <xsl:value-of select="name()"/>
                </span>
              </td>
              <td>
```

```
                <span class="literal">
                  <xsl:value-of select="."/>
                </span>
              </td>
            </tr>
          </xsl:for-each>
        </table>
      </td>
    </tr>
  </xsl:if>
  ...
```

To illustrate the structure of the document, we create a table for each element. If that element in turn contains other elements, we put those in separate tables as well. After creating the table, we check for any attribute nodes (`count(@*) > 0`). If there are any attributes, we create a new table to list them.

Once all the attributes have been processed, we look for all the namespace nodes and list them as well. To do this, we employ the rarely used `namespace` axis:

```
<xsl:for-each select="namespace::*">
  <xsl:if test="name() != 'xml'">
    <xsl:call-template name="namespace-node"/>
  </xsl:if>
</xsl:for-each>
```

We don't display the `xml` namespace prefix because it's always defined and always associated with the namespace URI `http://www.w3.org/XML/1998/namespace`. For a namespace node, the `name( )` function returns the namespace prefix. The value of a namespace node (`<xsl:value-of select="."/>`) is its URI. If we were looking for a namespace node that could be associated with any prefix, we would test the value of the namespace node instead. For example, this namespace definintion:

```
xmlns:sample="http://www.w3c.org/1999/XSL/Transform"
```

associates the XSLT namespace with the prefix `sample`. If we wanted to see whether the XSLT namespace were defined, we would have to use the value of the namespace node instead of simply looking for the usual prefix `xsl`.

Finally, we use `<xsl:apply-templates>` to process everything contained in the element:

```
<xsl:apply-templates select="node()"/>
```

Here's the complete stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!--xpath-1_0-tree-diagram.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="html"/>
```

```
<xsl:template match="/">
  <html>
    <head>
      <title>XPath view of your document</title>
      <style type="text/css">
        <xsl:comment>
          .literal   { font-family: Courier, monospace; }
          .docroot   { background-color: #99CCCC; }
          .element   { background-color: #CCCC99; }
          .attribute { background-color: #FFFF99; }
          .text      { background-color: #FFCC99; }
          .comment   { background-color: #CCCCFF; }
          .pi        { background-color: #99FF99; }
          .namespace { background-color: #CC99CC; }
          .box       { border: solid black 3px; }
        </xsl:comment>
      </style>
    </head>
    <body style="font-family: sans-serif;">
      <h1>XPath view of your document</h1>
      <p>
        The structure of your document (as XPath sees it)
        is outlined below.
      </p>
      <table cellspacing="5" cellpadding="2" border="0">
        <tr>
          <td colspan="7">
            <b>Node types:</b>
          </td>
        </tr>
        <tr>
          <td class="docroot"><b>document root</b></td>
          <td class="element"><b>element</b></td>
          <td class="attribute"><b>attribute</b></td>
          <td class="text"><b>text</b></td>
          <td class="comment"><b>comment</b></td>
          <td class="pi"><b>processing instruction</b></td>
          <td class="namespace"><b>namespace</b></td>
        </tr>
      </table>
      <br/>
      <table width="100%" class="box" bgcolor="#FFFFFF"
        title="document root" alt="document root">
        <tr class="docroot">
          <td colspan="3">
            <b>document root:</b>
          </td>
        </tr>
        <tr>
          <td width="15" class="docroot"></td>
          <td>
            <table width="100%">
              <xsl:apply-templates
                select="*|comment()|processing-instruction()|text()"/>
```

```
            </table>
          </td>
          <td width="15" class="docroot"></td>
        </tr>
        <tr class="docroot">
          <td colspan="3"> </td>
        </tr>
      </table>
    </body>
  </html>
</xsl:template>

<xsl:template match="comment()">
  <tr>
    <td class="comment">
      <b>comment: </b>
      <span class="literal">
        <xsl:value-of select="."/>
      </span>
    </td>
  </tr>
</xsl:template>

<xsl:template match="processing-instruction()">
  <tr>
    <td class="pi">
      <b>processing instruction: </b>
      <span class="literal">
        <xsl:text>&lt;?</xsl:text>
        <xsl:value-of select="name()"/>
        <xsl:text>?&gt;</xsl:text>
        <br/>
        <xsl:value-of select="."/>
      </span>
    </td>
  </tr>
</xsl:template>

<xsl:template match="text()">
  <xsl:if test="string-length(normalize-space(.))">
    <tr>
      <td class="text" width="100%">
        <b>text: </b>
        <span class="literal">
          <xsl:value-of select="."/>
        </span>
      </td>
    </tr>
  </xsl:if>
</xsl:template>

<xsl:template name="namespace-node">
  <tr>
    <td class="namespace">
      <b>namespace: </b>
```

```
      <span class="literal">
        <xsl:value-of select="name()"/>
      </span>
      <br/>
      <span class="literal">
        <xsl:value-of select="."/>
      </span>
    </td>
  </tr>
</xsl:template>

<xsl:template match="*">
  <xsl:variable name="title">
    <xsl:for-each select="ancestor-or-self::*">
      <xsl:text>/</xsl:text>
      <xsl:value-of select="name()"/>
    </xsl:for-each>
  </xsl:variable>
  <tr title="{$title}" alt="{$title}">
    <td>
      <table class="box" width="100%">
        <tr>
          <td class="element" colspan="3" valign="top">
            <b>element: </b>
            <span class="literal">
              <xsl:text>&lt;</xsl:text>
              <xsl:value-of select="name()"/>
              <xsl:text>&gt;</xsl:text>
            </span>
          </td>
        </tr>
        <tr>
          <td class="element" width="15">  </td>
          <td>
            <table border="0" width="100%">
              <xsl:if test="count(@*) &gt; 0">
                <tr>
                  <td>
                    <table width="100%">
                      <tr class="attribute">
                        <td width="20%">
                          <b>attribute name</b>
                        </td>
                        <td>
                          <b>value</b>
                        </td>
                      </tr>
                      <xsl:for-each select="@*">
                        <tr class="attribute">
                          <td width="20%">
                            <span class="literal">
                              <xsl:value-of select="name()"/>
                            </span>
                          </td>
                          <td>
```

```
                        <span class="literal">
                          <xsl:value-of select="."/>
                        </span>
                      </td>
                    </tr>
                  </xsl:for-each>
                </table>
              </td>
            </tr>
          </xsl:if>
          <xsl:for-each select="namespace::*">
            <xsl:if test="name() != 'xml'">
              <xsl:call-template name="namespace-node"/>
            </xsl:if>
          </xsl:for-each>
          <xsl:apply-templates select="node()"/>
        </table>
      </td>
      <td bgcolor="#CCCC99" width="15"> </td>
    </tr>
    <tr>
      <td colspan="3" bgcolor="#CCCC99"> </td>
    </tr>
  </table>
 </td>
</tr>
</xsl:template>

</xsl:stylesheet>
```

Before we leave this example, a couple of other techniques are worth mentioning here. First, notice that we used CSS to format some of the output. XSLT and CSS aren't mutually exclusive; you can use XSLT to generate CSS as part of an HTML page, as we demonstrated here. Second, we used wildcard expressions such as * and `@*` to process all the elements and attributes in our document. Use of these expressions allows us to apply this stylesheet to any XML document, regardless of the tags it uses. Because we use these wildcard expressions, we have to use the `name( )` function to get the name of the element or attribute we're currently working with. Third, notice that we used conditional logic and the expression `count(@*) > 0` to determine whether a given element has attributes. We'll talk more about conditional logic in Chapter 5.

Summary

We've covered the basics of XPath. Hopefully, at this point you're comfortable with the idea of writing XPath expressions to describe parts of an XML document. As we go through the following chapters, you'll see XPath expressions used in a variety of ways, all of which build on the basics we've discussed here. When you're debugging a stylesheet, you'll probably spend most of your time making sure your XPath expressions select the right data. Very few of the things we'll do in the rest of the book are possible without precise XPath expressions.

CHAPTER 4

Creating Output

Goals of This Chapter

By the end of this chapter, you should know how to:

- Generate text
- Number things, including numbering at multiple levels
- Format numbers
- *[2.0]* Format dates and times
- Use `<xsl:copy>` and `<xsl:copy-of>` to copy nodes from the input document to the output document
- Deal with whitespace

Generating Text

The first thing we'll cover is how to put text in the output. Just putting some text out there is simple enough, but we'll look at some more advanced techniques that we'll explore throughout the book. We'll look at two elements in particular: `<xsl:text>` and `<xsl:value-of>`. We'll use these to create an HTML document that contains a table of contents for an XML document. Here's the XML document we'll use:

```
<?xml version="1.0"?>
<!-- toc_source.xml -->
<article>
  <title>Creating output</title>
  <body>
    <heading1>Generating text</heading1>
    <heading1>Numbering things</heading1>
    <heading1>Formatting numbers</heading1>
    <heading1>Copying nodes from the input document to the output</heading1>
    <heading1>Handling whitespace</heading1>
  </body>
</article>
```

Creating Simple Text

There are many times you need to write some text to the output. In the first example we'll build in this chapter, we want to create HTML that looks like this:

```
<h1>Table of Contents</h1>
<h2>Generating text</h2>
<h2>Numbering things</h2>
<h2>Formatting numbers</h2>
<h2>Copying nodes from the input document to the output</h2>
<h2>Handling whitespace</h2>
```

In this output document, the text of each item in the table is the text of a particular element in the XML source. The text `Table of Contents`, however, is the same each time. To generate this text, we'll use the `<xsl:text>` element. We'll start our stylesheet by generating that text:

```
<xsl:template match="/">
  <h1>
    <xsl:text>Table of Contents</xsl:text>
  </h1>
  ...
</xsl:template>
```

All we did here was insert a string in our output document. To make things even simpler, we could have done this:

```
<xsl:template match="/">
  <h1>Table of Contents</h1>
  ...
</xsl:template>
```

For any non-XSLT element in a stylesheet (`<h1>`, for example), XSLT's default behavior is to simply pass that element to the output. Normally we'll use `<xsl:text>` when we need complete control over whitespace or when we're creating text output instead of a marked-up document such as an HTML file.

In these examples, we simply wrote text to our output document; most often we'll combine text with values from our source document. For example, we might want to generate an HTML document that looks like this:

```
<h1>Table of Contents</h1>
<p>This document contains <b>5</b> chapters:</p>
<h2>Generating text</h2>
<h2>Numbering things</h2>
...
```

The paragraph we added contains text as well as a value, which is the number of chapters in our source document. To output values such as this or the title of each chapter we need the `<xsl:value-of>` element, which we'll look at in just a minute.

Before we move on to the <xsl:value-of> element, a quick note: if we were writing Java or C# code to create the output document, we wouldn't put literal text such as Table of Contents in the code itself. We'd load that string at runtime from a file of translated strings. That would let us use the same code to create a table of contents in English, Japanese, Polish, or whatever language we need. Setting up stylesheets to do the same thing is complicated, but we'll discuss how to do just that later in "The document() Function" in Chapter 8.

Outputting the Value of Something

Creating a simple text string is easy: we just use the <xsl:text> element. However, in any stylesheet you write, you'll probably need to output the value of something from the XML source. That's what the <xsl:value-of> element is for. In the example stylesheet we're building, we want to transform this element in the XML source:

```
<heading1>Generating text</heading1>
```

into this HTML element in the output:

```
<h2>Generating text</h2>
```

In other words, we want to take every <heading1> element and transform it into an HTML <h2> element that contains the value of the <heading1> element. Here's a template that does just that:

```
<xsl:template match="heading1">
  <h2>
    <xsl:value-of select="."/>
  </h2>
</xsl:template>
```

To generate our earlier paragraph that contains the number of chapters, we'll use <xsl:value-of> with the XPath count( ) function:

```
<p>
  This document contains
  <xsl:value-of select="count(/article/body/heading1)"/>
  chapters.
</p>
```

This bit of XSLT creates the following HTML paragraph:

```
<p>
   This document contains
   5
   chapters.

</p>
```

An HTML browser normalizes the whitespace before rendering it, as shown in Figure 4-1.

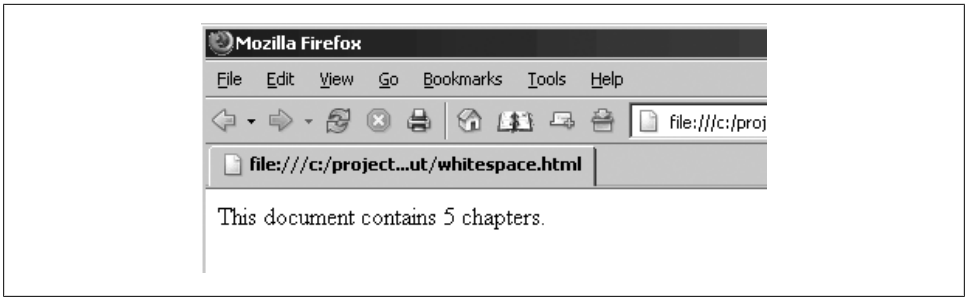


Figure 4-1. HTML normalizes whitespace before displaying a document

If we needed more control over the output, we could use `<xsl:value-of>` and `<xsl:text>` together:

```
<p>
  <xsl:text>This document contains </xsl:text>
  <xsl:value-of select="count(/article/body/heading1)"/>
  <xsl:text> chapters.</xsl:text>
</p>
```

This generates an HTML paragraph without any extra whitespace:

```
<p>This document contains 5 chapters.</p>
```

Notice that we had to put blank spaces inside the `<xsl:text>` elements so there would be spaces around the number 5. Finally, a simpler, but far less readable alternative would be to run all of the text and the `<xsl:value-of>` element together on a single line:

```
<p>This document contains <xsl:value-of select="count(/article/body...
```

HTML browsers typically handle whitespace the way you want; for other types of output, dealing with whitespace is a significant issue. We'll talk more about whitespace a little later.

[2.0] Changes to <xsl:value-of> in XSLT 2.0

In XSLT 2.0, `<xsl:value-of>` has a `separator` attribute. If the `select` attribute is a sequence of items, those items are output in sequence, with the value of the `separator` attribute between them. We'll look at a couple of examples that use `separator` here.

First of all, here's a short XML document that lists different automobile manufacturers and some of the cars they make:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- cars.xml -->
<cars>
  <manufacturer name="Chevrolet">
    <car>Cavalier</car>
    <car>Corvette</car>
    <car>Impala</car>
    <car>Malibu</car>
  </manufacturer>
```

```
  <manufacturer name="Ford">
    <car>Pinto</car>
    <car>Mustang</car>
    <car>Taurus</car>
  </manufacturer>
  <manufacturer name="Volkswagen">
    <car>Beetle</car>
    <car>Jetta</car>
    <car>Passat</car>
    <car>Touraeg</car>
  </manufacturer>
</cars>
```

Our first `<xsl:value-of>` element uses the `separator` attribute to list all of the manufacturers, separated by commas:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- value-of_2_0.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:value-of select="cars/manufacturer/@name" separator=", "/>
  </xsl:template>

</xsl:stylesheet>
```

Transforming our XML document with this stylesheet gives us the following results:

```
Chevrolet, Ford, Volkswagen
```

The `separator` attribute is useful in this case because it is inserted after every value except the last. In XSLT 1.0, we have to do something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- value-of_1_0.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:for-each select="cars/manufacturer/@name">
      <xsl:value-of select="."/>
      <xsl:if test="not(position()=last())">
        <xsl:text>, </xsl:text>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

After we output each item, we have to see whether that item is the last; if not, we insert our separator with `<xsl:text>`. The `<xsl:for-each>` and `<xsl:if>` elements aren't

necessary in XSLT 2.0; the `separator` attribute makes it much easier to create the output we want.

We can use the `separator` attribute with sequences as well. Here's an example that uses a sequence of `xs:string`s and a sequence created with XPath 2.0's new `to` operator:

```
<?xml version="1.0"?>
<!-- value-of_sequences.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:variable name="months" as="xs:string*"
    select="'January', 'February', 'March', 'April',
            'May', 'June', 'July', 'August',
            'September', 'October', 'November', 'December'"/>

  <xsl:template match="/">
    <xsl:value-of select="1 to 7" separator=", "/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="$months" separator="&#xA;"/>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates the following results:

```
1, 2, 3, 4, 5, 6, 7
January
February
March
April
May
June
July
August
September
October
November
December
```

Numbering Things

XSLT provides the `<xsl:number>` element to number the parts of a document. (It can also be used to format a numeric value; more on that later.) In general, `<xsl:number>` counts something. We'll look at a variety of examples here.

To fully illustrate how `<xsl:number>` works, we'll need an XML document with some things to count. We'll reuse our list of cars from the previous section:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- cars.xml -->
```

```
<cars>
  <manufacturer name="Chevrolet">
    <car>Cavalier</car>
    <car>Corvette</car>
    <car>Impala</car>
    <car>Malibu</car>
  </manufacturer>
  <manufacturer name="Ford">
    <car>Pinto</car>
    <car>Mustang</car>
    <car>Taurus</car>
  </manufacturer>
  <manufacturer name="Volkswagen">
    <car>Beetle</car>
    <car>Jetta</car>
    <car>Passat</car>
    <car>Touraeg</car>
  </manufacturer>
</cars>
```

We'll use `<xsl:number>` in several different ways to illustrate the various options we have in numbering things. We'll start with something simple:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- number1.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Automobile manufacturers and their cars</title>
      </head>
      <body>
        <xsl:for-each select="cars/manufacturer">
          <p>
            <xsl:number format="1. "/>
            <xsl:value-of select="@name"/>
          </p>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

We get this HTML document:

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Automobile manufacturers and their cars</title>
   </head>
   <body>
```

```
      <p>1. Chevrolet</p>
      <p>2. Ford</p>
      <p>3. Volkswagen</p>
   </body>
</html>
```

This is about the simplest example of `<xsl:number>` that you can write. (You could leave off the `format` attribute, but you'd get paragraphs such as `<p>1Chevrolet</p>`—probably not what you want.) Changing the stylesheet to use `format="a. "` generates these paragraphs:

```
      <p>a. Chevrolet</p>
      <p>b. Ford</p>
      <p>c. Volkswagen</p>
```

Here's what we get with `format="i. "`:

```
      <p>i. Chevrolet</p>
      <p>ii. Ford</p>
      <p>iii. Volkswagen</p>
```

The `<xsl:number>` element has lots of other attributes and capabilities; we'll look at the most common ones here. (See the complete description of the `<xsl:number>` element in Appendix A.) Here's an example that uses the `value` attribute:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- number2.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Automobile manufacturers and their cars</title>
      </head>
      <body>
        <xsl:for-each select="cars/manufacturer">
          <p>
            <xsl:text>Cars produced by </xsl:text>
            <xsl:value-of select="@name"/>
            <xsl:text>: </xsl:text>
            <xsl:number value="count(car)" format="01"/>
          </p>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates this HTML document:

```
<html>
   <head>
```

```
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Automobile manufacturers and their cars</title>
    </head>
    <body>
      <p>Cars produced by Chevrolet: 04</p>
      <p>Cars produced by Ford: 03</p>
      <p>Cars produced by Volkswagen: 04</p>
    </body>
</html>
```

In this case, we could have used `<xsl:value-of select="count(car)"/>` to get similar results, but `<xsl:number>` lets us format the number as we want.

Using `<xsl:number>` with the `format` attribute is a good way to format any value, whether it comes from the XML source or not. For example, this markup:

```
<xsl:number value="1965" format="I"/>
```

produces the text `MCMLXV`.

As you'd expect, there are more powerful things we can do. Here's an example that uses the `level` and `count` attributes:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- number3.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Automobile manufacturers and their cars&#xA;</xsl:text>
    <xsl:for-each select="cars/manufacturer">
      <xsl:number count="manufacturer" format="1. "/>
      <xsl:value-of select="@name"/>
      <xsl:text>&#xA;</xsl:text>
      <xsl:for-each select="car">
        <xsl:number count="manufacturer|car" level="multiple"
          format="1.1. "/>
        <xsl:value-of select="."/>
        <xsl:text>&#xA;</xsl:text>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet gives us the following text document:

```
Automobile manufacturers and their cars
1. Chevrolet
1.1. Cavalier
1.2. Corvette
1.3. Impala
1.4. Malibu
2. Ford
2.1. Pinto
```

```
2.2. Mustang
2.3. Taurus
3. Volkswagen
3.1. Beetle
3.2. Jetta
3.3. Passat
3.4. Touraeg
```

The `count` attribute tells the XSLT processor what elements to count, and `level="multiple"` counts the manufacturers at one level and the cars per manufacturer at another. Notice that in the second `<xsl:for-each<` element we used the attribute `count="manufacturer|car"`, even though we're looking only at `<car>` elements. That's because the number `3.2` means the second `<car>` from the third `<manufacturer>`. If we don't include the manufacturers in our count, we won't get the results we want.

The values for `level` are `single`, `multiple`, and `any`. The value `single`, the default, counts only an item's siblings, while `multiple` counts an item along with any of its ancestors (that's what we did in the previous example). `level="any"` counts an item along with everything that occurred before it in the document, whether it's an ancestor of the current item or not. Here's an example that uses `level="any"`:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- number4.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Automobile manufacturers and their cars&#xA;</xsl:text>
    <xsl:for-each select="cars/manufacturer">
      <xsl:number count="manufacturer|car" level="any" format="1. "/>
      <xsl:value-of select="@name"/>
      <xsl:text>&#xA;</xsl:text>
      <xsl:for-each select="car">
        <xsl:number count="manufacturer|car" level="any" format="1. "/>
        <xsl:value-of select="."/>
        <xsl:text>&#xA;</xsl:text>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Here we use an identical `<xsl:number>` element in both places. We're counting all of the `<manufacturer>` and `<car>` elements, so we need to use `level="any"` and `count="manufacturer|car"` in both places. Here are the results:

```
Automobile manufacturers and their cars
1. Chevrolet
2. Cavalier
3. Corvette
4. Impala
5. Malibu
```

```
6. Ford
7. Pinto
8. Mustang
9. Taurus
10. Volkswagen
11. Beetle
12. Jetta
13. Passat
14. Touraeg
```

Also keep in mind that you can use `<xsl:number>` at isolated times. Here's a contrived example that counts only even-numbered cars from each manufacturer:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- number5.xsl -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Automobile manufacturers and their cars&#xA;</xsl:text>
    <xsl:for-each select="cars/manufacturer">
      <xsl:value-of select="@name"/>
      <xsl:text>&#xA;</xsl:text>
      <xsl:for-each select="car">
        <xsl:text>  </xsl:text>
        <xsl:if test="(position() mod 2) = 0">
          <xsl:number count="manufacturer|car" level="multiple"
            format="1.1. "/>
        </xsl:if>
        <xsl:value-of select="."/>
        <xsl:text>&#xA;</xsl:text>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

To select only even-numbered items, we use the XPath `mod` operator. If we divide the position of the current element by 2 and the result is zero, we know we have an even-numbered item. This stylesheet gives us the following list:

```
Automobile manufacturers and their cars
Chevrolet
  Cavalier
  1.2. Corvette
  Impala
  1.4. Malibu
Ford
  Pinto
  2.2. Mustang
  Taurus
Volkswagen
  Beetle
  3.2. Jetta
```

```
  Passat
  3.4. Touraeg
```

In this example, we used <xsl:number> only for even-numbered cars from each manufacturer, and we never used <xsl:number> for the <manufacturer> element at all. Despite that, the <xsl:number> element calculates the correct value based on the position of the current item in the source document.

That can lead to some complications, however. If we sort the source document as we process it, our numbers look a little strange. We'll add a <xsl:sort> element to our stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- number6.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Automobile manufacturers and their cars&#xA;</xsl:text>
    <xsl:for-each select="cars/manufacturer">
      <xsl:value-of select="@name"/>
      <xsl:text>&#xA;</xsl:text>
      <xsl:for-each select="car">
        <xsl:sort select="."/>
        <xsl:text>  </xsl:text>
        <xsl:if test="(position() mod 2) = 0">
          <xsl:number count="manufacturer|car" level="multiple"
            format="1.1. "/>
        </xsl:if>
        <xsl:value-of select="."/>
        <xsl:text>&#xA;</xsl:text>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

And our results look like this:

```
Automobile manufacturers and their cars
Chevrolet
  Cavalier
  1.2. Corvette
  Impala
  1.4. Malibu
Ford
  Mustang
  2.1. Pinto
  Taurus
Volkswagen
  Beetle
  3.2. Jetta
```

```
  Passat
  3.4. Touraeg
```

The number for `Pinto` isn't what we expected. That's because the `position( )` function is based on the current (sorted) context, while the numbering we're doing is based on the original document order. In the sorted order `Pinto` is the second item, so the `test` of our `<xsl:if>` element is true. When we use `<xsl:number>`, however, `Pinto` is the first `<car>` in the source document. The result is the number `2.1` instead of the `2.2` we expected.

It's not pretty, but here's a stylesheet that fixes the problem. We use `<xsl:number>` to count `<manufacturer>` elements; that generates the first part of the number. Next we use `position( )` to output the position of the *sorted* element. The stylesheet looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- number7.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Automobile manufacturers and their cars&#xA;</xsl:text>
    <xsl:for-each select="cars/manufacturer">
      <xsl:value-of select="@name"/>
      <xsl:text>&#xA;</xsl:text>
      <xsl:for-each select="car">
        <xsl:sort select="."/>
        <xsl:text>  </xsl:text>
        <xsl:if test="(position() mod 2) = 0">
          <xsl:number count="manufacturer" level="multiple"
            format="1."/>
          <xsl:value-of select="position()"/>
          <xsl:text>. </xsl:text>
        </xsl:if>
        <xsl:value-of select="."/>
        <xsl:text>&#xA;</xsl:text>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

This numbers the `Pinto` using the sorted order:

```
Automobile manufacturers and their cars
Chevrolet
  Cavalier
  1.2. Corvette
  Impala
  1.4. Malibu
Ford
  Mustang
  2.2. Pinto
```

```
  Taurus
Volkswagen
  Beetle
  3.2. Jetta
  Passat
  3.4. Touraeg
```

We used `<xsl:number>` to count the position of the current car within the current manufacturer.

[2.0] Changes to <xsl:number> in XSLT 2.0

There are some changes to the `<xsl:number>` element in XSLT 2.0. First of all, three new formats have been added: `w`, `W`, and `Ww`. Those formats produce words in the default language on your machine (there's also a `lang` attribute you can use to change the language). Here's an example:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Automobile manufacturers and their cars&#xA;</xsl:text>
    <xsl:for-each select="cars/manufacturer">
      <xsl:value-of select="@name"/>
      <xsl:text>&#xA;</xsl:text>
      <xsl:for-each select="car">
        <xsl:text>  Car </xsl:text>
        <xsl:number count="car" level="single" format="w"/>
        <xsl:text>: </xsl:text>
        <xsl:value-of select="."/>
        <xsl:text>&#xA;</xsl:text>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet produces this text:

```
Automobile manufacturers and their cars
Chevrolet
  Car one: Cavalier
  Car two: Corvette
  Car three: Impala
  Car four: Malibu
Ford
  Car one: Pinto
  Car two: Mustang
  Car three: Taurus
Volkswagen
  Car one: Beetle
```

```
Car two: Jetta
Car three: Passat
Car four: Touraeg
```

Using the format `W` produces the numbers `ONE`, `TWO`, `THREE`, and so forth, while the format `Ww` produces `One`, `Two`, `Three`.

Another difference is the addition of the `select` attribute. Normally `<xsl:number>` numbers the current node; you can use the `select` attribute to generate the number for another node.

XSLT 2.0 also adds the `ordinal` attribute; `ordinal="yes"` combined with `format="1"` generates `1st`, `2nd`, `3rd`, while `ordinal="yes"` combined with `format="Ww"` generates `First`, `Second`, `Third`. The `ordinal` attribute has many different options that depend on the `lang` attribute and the `format` attribute; as you would expect, each XSLT 2.0 processor supports a different set of languages and options for the `ordinal` attribute. See your processor's documentation for information on what capabilities are available.

Finally, XSLT 2.0 signals an error if your `<xsl:number>` element has incompatible attributes. For example, you can't have a `select` attribute and a `value` attribute. In XSLT 1.0, the extra attributes were ignored.

Appendix F has a complete description of all the formatting codes for numbers, dates, times, and durations.

Formatting Decimal Numbers

We've already seen several ways of formatting decimal numbers using `<xsl:number>`. However, if we're going to work with numbers, we'll almost certainly have to deal with decimals. XSLT defines the `format-number( )` function and the `<xsl:decimal-format>` element to do just that. We'll use `<xsl:decimal-format>` to define a pattern for formatting numbers, and then we'll use `format-number( )` to apply a pattern to a number.

This stylesheet has several examples of `<xsl:decimal-format>` and `format-number( )`:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- decimal-format.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <!-- This format has no name, so it's assumed to be the default. -->
  <xsl:decimal-format decimal-separator="," grouping-separator="."/>

  <xsl:decimal-format name="us_default"/>

  <xsl:decimal-format name="other_options" NaN="[not a number]"
```

```
    infinity="unfathomably huge"/>

  <xsl:decimal-format name="hash_mark" digit="!"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of &lt;xsl:decimal-format&gt; and </xsl:text>
    <xsl:text>format-number():</xsl:text>
    <xsl:text>&#xA;&#xA;  1. format-number(3728493.3882, </xsl:text>
    <xsl:text>'#.###,##') : </xsl:text>
    <xsl:value-of
      select="format-number(3728493.3882, '#.###,##')"/>

    <xsl:text>&#xA;&#xA;  2. format-number(3728493.3882, </xsl:text>
    <xsl:text>'#,###.##', 'us_default') : </xsl:text>
    <xsl:value-of
      select="format-number(3728493.3882, '#,###.##', 'us_default')"/>

    <xsl:text>&#xA;&#xA;  3. format-number(number(1) div 0, '#.#') : </xsl:text>
    <xsl:value-of select="format-number(number(1) div 0, '#.#')"/>

    <xsl:text>&#xA;&#xA;  4. format-number(number(1) div 0, '#.#', </xsl:text>
    <xsl:text>'other_options') : &#xA;       </xsl:text>
    <xsl:value-of
      select="format-number(number(1) div 0, '#.#', 'other_options')"/>

    <xsl:text>&#xA;&#xA;  5. format-number(number('blue') * </xsl:text>
    <xsl:text>number('orange'), '#') : </xsl:text>
    <xsl:value-of
      select="format-number(number('blue') * number('orange'), '#')"/>

    <xsl:text>&#xA;&#xA;  6. format-number(number('blue') * </xsl:text>
    <xsl:text>number('orange'), '#', 'other_options') : </xsl:text>
    <xsl:text>&#xA;       </xsl:text>
    <xsl:value-of
      select="format-number(number('blue') * number('orange'), '#',
              'other_options')"/>

    <xsl:text>&#xA;&#xA;  7. format-number(42, '#!', </xsl:text>
    <xsl:text>'hash_mark') : </xsl:text>
    <xsl:value-of select="format-number(42, '#!', 'hash_mark')"/>
  </xsl:template>

</xsl:stylesheet>
```

When we run this stylesheet against any document, we get this output:

```
Tests of <xsl:decimal-format> and format-number():

  1. format-number(3728493.3882, '#.###,##') : 3.728.493,39

  2. format-number(3728493.3882, '#,###.##', 'us_default') : 3,728,493.39

  3. format-number(number(1) div 0, '#.#') : Infinity

  4. format-number(number(1) div 0, '#.#', 'other_options') :
       unfathomably huge
```

```
5. format-number(number('blue') * number('orange'), '#') : NaN

6. format-number(number('blue') * number('orange'), '#', 'other_options') :
     [not a number]

7. format-number(42, '#!', 'hash_mark') : #42
```

This is an XSLT 1.0 stylesheet, but changing the `version` attribute to `2.0` generates the same results. We'll discuss each line in the output and point out some differences in the way XSLT 2.0 processes numbers as we go.

1. This example formats a number using the default format we defined. Because our stylesheet has a `<xsl:decimal-format>` element without a name, this format is used unless the name of another `<xsl:decimal-format>` element is used. We defined the default format to use periods as the thousands separator and a comma as the decimal point. Notice that to get this to work we had to use the period and comma appropriately in the formatting string. The numeric value itself uses the decimal point as usual.
2. This is the same example as before, only we pass the name of a number format as the third argument. Notice that the decimal format `us_default` doesn't have any attributes; that means it uses a period as the decimal point and a comma as the thousands separator.
3. This generates the default value for infinity, which is the string `Infinity`. In XSLT 1.0, the expression `1 div 0` generates the same result.

   [2.0] In XSLT 2.0, any value that is an `xs:integer` or `xs:decimal` cannot have the value infinity, so `1 div 0` won't run at all. Calling the function `number(1)` converts `1` into the `xs:double` equivalent, so `number(1) div 0` works. (That's a lot of work to divide a number by zero, but it does explain some of the details of how math works in XSLT 2.0.)
4. This generates the value for infinity defined in the number format `other_options`. The same details apply here as in the previous example; the expression `1 div 0` doesn't work in XSLT 2.0.
5. This generates `NaN`. In XSLT 1.0, the expression `'blue' * 'orange'` generates the same result.

   [2.0] In XSLT 2.0, the expression `'blue' * 'orange'` doesn't work because the multiplication symbol requires two numbers. Using the `number( )` function turns each of the strings into the numeric value `NaN`. To generate `NaN` in XSLT 2.0, `number('NaN')` does the trick, as do `number('blue')`, `number('orange')`, and `number('any old string at all')`.
6. This generates `[not a number]`, the value defined in the number format `other_options`. (The same restrictions for XSLT 2.0 apply here as well.)

7. This generates `#42`. This uses the number format `hash_mark`, which defines an exclamation point as the digit character in the picture string. This allows us to put the hash mark into the picture string.

[2.0] Formatting Dates and Times

XSLT 2.0 adds three new formatting functions, `format-date( )`, `format-time( )`, and `format-dateTime( )`. We'll take `xs:date`, `xs:time`, and `xs:dateTime` values and format them in some useful way.

You can call each of these functions in two ways. The simplest is to pass the function a value and a formatting string. If you need more detail, the second way of calling these functions requires you to specify a language, a calendar, and a country as well. The XSLT 2.0 specification lists more than 25 different calendars used around the world, and there are hundreds of combinations of language and country codes. Look at the documentation for your XSLT processor to see which calendars, languages, and countries it supports.

Our first example is pretty simple. We'll create a stylesheet that uses the XPath functions `current-date( )`, `current-time( )`, and `current-dateTime( )`:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- datetime1.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of date and time formatting:&#xA;</xsl:text>
    <xsl:text>&#xA;  The current date is </xsl:text>
    <xsl:value-of select="format-date(current-date(),
                          '[M01]/[D01]/[Y0001]')"/>
    <xsl:text>&#xA;  The current time is </xsl:text>
    <xsl:value-of select="format-time(current-time(),
                          '[H01]:[m01] [z]')"/>
    <xsl:text>&#xA;  It's currently </xsl:text>
    <xsl:value-of select="format-dateTime(current-dateTime(),
                          '[h1]:[m01] [P] on [MNn] [D].')"/>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet produces this text:

```
Tests of date and time formatting:

  The current date is 03/08/2006
  The current time is 22:27 GMT-5
  It's currently 10:27 p.m. on March 8.
```

In this stylesheet, `M01` produces the two-digit month, `D01` produces the two-digit day, and `Y0001` produces the four-digit year. `H01` produces the 2-digit hour in a 24-hour clock, `m01` produces the 2-digit minutes, `z` produces the time zone, `h1` produces the 1- or 2-digit hour in a 12-hour clock, and `P` generates `a.m.` or `p.m.` Finally, `MNn` generates the capitalized name of the month.

The formatting codes used by the `<xsl:number>` element can be used in the picture string for these functions. Here's another stylesheet that uses more of those formatting codes:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- datetime2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;More tests of date and time formatting:&#xA;</xsl:text>
    <xsl:text>&#xA;  Today is the </xsl:text>
    <xsl:value-of select="format-date(current-date(),
                          '[Dwo] day of [MNn], [Y0001]')"/>
    <xsl:text>&#xA;  Right now is the </xsl:text>
    <xsl:value-of select="format-time(current-time(),
                          '[m1o] minute of the [Hwo] hour of the day.')"/>
    <xsl:text>&#xA;  It's currently </xsl:text>
    <xsl:value-of select="format-dateTime(current-dateTime(),
                          '[h01]:[m01] [P] on [FNn] the [D1o].')"/>
    <xsl:text>&#xA;  Today is the </xsl:text>
    <xsl:value-of select="format-date(current-date(),
                          '[dwo]')"/>
    <xsl:text> day of the year. </xsl:text>
    <xsl:text>&#xA;  December 25, 1960 in German: </xsl:text>
    <xsl:value-of select="format-date(xs:date('1960-12-25'),
                          '[D] [MNn,3-3] [Y0001]', 'de',
                          'AD', 'DE')"/>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates this text:

```
More tests of date and time formatting:

  Today is the eighth day of March, 2006
  Right now is the 28th minute of the twenty-second hour of the day.
  It's currently 10:28 p.m. on Wednesday the 8th.
  Today is the sixty-seventh day of the year.
  December 25, 1960 in German: 25 Dez 1960
```

Here are the explanations for all the formatting codes in this example:

`Dwo`
: The word for the ordinal value of the day

`MNn`
: The capitalized name of the month

`Y0001`
: The four-digit year

`m1o`
: The numeric ordinal of the minute

`Hwo`
: The hour, expressed as an ordinal word

`h01`
: The 2-digit hour in a 24-hour clock

`m01`
: The 2-digit minute

`P`
: The a.m. or p.m. indicator

`FNn`
: The capitalized name of the day of the week

`D1o`
: The numeric ordinal of the day

`dwo`
: The day of the year, expressed as an ordinal word

`D`
: The numeric day of the month

`MNn,3-3`
: The capitalized name of the month, returned in a string that's at least three characters long, but no more than three characters long

For the last call to `format-date( )`, we used the five-option version of the function, passing in the language, calendar and country codes. The formatting codes defined by XSLT give you a wide range of options for formatting the different components of dates and times.

Using <xsl:copy> and <xsl:copy-of>

As you transform your XML input document into something else, you'll often want to just copy a given element to the output document. XSLT provides two elements that do this: `<xsl:copy>` and `<xsl:copy-of>`. We'll discuss them here and go through several stylesheets that use these elements to create output.

A Stylesheet That Reproduces Its Input Document

To start our examples, we'll look at a stylesheet that generates a document equal to the input document. (This is sometimes called an *identity transform*.) The stylesheet is short and sweet:

```
<?xml version="1.0"?>
<!-- copy-of.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:copy-of select="*"/>
  </xsl:template>
</xsl:stylesheet>
```

That's all we have to do. Our template simply says to start with the document element of the input document and copy it to the output. `<xsl:copy-of>` does a *deep copy* of a node, so the root node and all of its children are copied to the output. If any of the root node's descendants are element nodes with attributes, the attributes are copied as well. (Remember, an element's attributes aren't considered children.)

We'll test our stylesheet against this document:

```
<?xml version="1.0"?>
<!-- sonnet.xml -->
<sonnet type='Shakespearean'>
  <auth:author xmlns:auth="http://www.authors.com/">
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
    <year-of-death>1616</year-of-death>
  </auth:author>
  <!-- Is there an official title for this sonnet?  They're
       sometimes named after the first line.                    -->
  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
    <line>If hairs be wires, black wires grow on her head.</line>
    <line>I have seen roses damasked, red and white,</line>
    <line>But no such roses see I in her cheeks.</line>
    <line>And in some perfumes is there more delight</line>
    <line>Than in the breath that from my mistress reeks.</line>
    <line>I love to hear her speak, yet well I know</line>
    <line>That music hath a far more pleasing sound.</line>
    <line>I grant I never saw a goddess go,</line>
    <line>My mistress when she walks, treads on the ground.</line>
    <line>And yet, by Heaven, I think my love as rare</line>
    <line>As any she belied with false compare.</line>
  </lines>
</sonnet>
```

The results look like this:

```
<?xml version="1.0" encoding="UTF-8"?><!-- sonnet.xml --><sonnet type="Shakespearean">
  <auth:author xmlns:auth="http://www.authors.com/">
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
    <year-of-death>1616</year-of-death>
  </auth:author>
  <!-- Is there an official title for this sonnet?  They're
       sometimes named after the first line.                    -->
  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
    <line>If hairs be wires, black wires grow on her head.</line>
    <line>I have seen roses damasked, red and white,</line>
    <line>But no such roses see I in her cheeks.</line>
    <line>And in some perfumes is there more delight</line>
    <line>Than in the breath that from my mistress reeks.</line>
    <line>I love to hear her speak, yet well I know</line>
    <line>That music hath a far more pleasing sound.</line>
    <line>I grant I never saw a goddess go,</line>
    <line>My mistress when she walks, treads on the ground.</line>
    <line>And yet, by Heaven, I think my love as rare</line>
    <line>As any she belied with false compare.</line>
  </lines>
</sonnet>
```

The result document looks almost exactly like the original. The `<sonnet>` element no longer begins on a separate line, and the XML declaration now includes `encoding="UTF-8"`. Also notice that the single quotes around the `type` attribute are now double quotes. These are changes from the text of the original XML document, but semantically the two are the same. Notice four things in particular that were copied to the output document:

- The comment before the document element
- The `type` attribute of the `<sonnet>` element
- The comment in the middle of the document
- The namespace declaration on the `<auth:author>` element

Earlier in our discussion we made a point of talking about the root node rather than the document element. The root node here contains two things: the comment outside the document element and the document element itself. Everything in the XML source is a descendant of the root node; the document element isn't always the root node's only child.

We'll look at the <xsl:copy> element next and see how it handles (or doesn't handle) our input document.

A Stylesheet That Doesn't Quite Reproduce Its Input Document

Now we'll look at a similar stylesheet that uses <xsl:copy> instead. Again, our stylesheet is very simple:

```
<?xml version="1.0"?>
<!-- copy1.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:copy/>
  </xsl:template>
</xsl:stylesheet>
```

You probably noticed that the stylesheet is shorter. Unlike <xsl:copy-of>, the <xsl:copy> element doesn't have a select attribute. (It has some other attributes in XSLT 2.0; it didn't have any at all in XSLT 1.0.) Here are the results when we use this stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Hmmm. It appears that <xsl:copy> didn't actually copy anything. While that makes for a small, concise document, it's probably not what we wanted. (This is the result you get from Xalan; Saxon doesn't generate anything at all.) One thing to remember here is that the document root is *not* the root element that contains the XML data in our document. An XML document can contain comments and processing instructions that are outside the root element; those comments and PIs are part of the XPath document root. So, if we want to copy anything, we need to create a template for the root element. Here's another attempt at a stylesheet:

```
<?xml version="1.0"?>
<!-- copy2.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:apply-templates select="*"/>
  </xsl:template>

  <xsl:template match="*">
    <xsl:copy/>
  </xsl:template>
</xsl:stylesheet>
```

And here are the results we get:

```
<?xml version="1.0" encoding="UTF-8"?><sonnet/>
```

Well, at least we have the <sonnet> element, but we don't have any of its children. We also lost the type attribute of the <sonnet> element. Using <xsl:copy> to copy our document requires using the <xsl:for-each> element to copy all the attributes of each element we're copying. Here's how the latest iteration of our stylesheet looks:

```
<?xml version="1.0"?>
<!-- copy3.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:apply-templates select="*"/>
  </xsl:template>

  <xsl:template match="*">
    <xsl:copy>
      <xsl:for-each select="@*">
        <xsl:copy/>
      </xsl:for-each>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

We used <xsl:for-each> to copy all of the attributes that an element might have. This stylesheet comes pretty close to duplicating the input document:

```
<?xml version="1.0" encoding="UTF-8"?><sonnet type="Shakespearean">
  <auth:author xmlns:auth="http://www.authors.com/">
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
    <year-of-death>1616</year-of-death>
  </auth:author>

  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
    <line>If hairs be wires, black wires grow on her head.</line>
    <line>I have seen roses damasked, red and white,</line>
    <line>But no such roses see I in her cheeks.</line>
    <line>And in some perfumes is there more delight</line>
    <line>Than in the breath that from my mistress reeks.</line>
    <line>I love to hear her speak, yet well I know</line>
    <line>That music hath a far more pleasing sound.</line>
    <line>I grant I never saw a goddess go,</line>
    <line>My mistress when she walks, treads on the ground.</line>
    <line>And yet, by Heaven, I think my love as rare</line>
    <line>As any she belied with false compare.</line>
  </lines>
</sonnet>
```

Even this version of the stylesheet doesn't copy comments or processing instructions. Notice that the output has a blank line in place of the comment in the original document. If we wanted to handle comments and processing instructions, we'd have to change the `match` and `select` attributes to make sure they were processed.

The point of these examples is that `<xsl:copy>` *forces you do to most of the work yourself*. You have complete control over what exactly gets copied, but that comes at a price. If you wanted only to copy certain elements, such as all `<customer>` elements with an `<address>` element in which the `<province>` element has a value of `PEI`, `<xsl:copy>` gives you the control to do that.

Before we go on to more complicated examples that use `<xsl:copy>`, be aware that we could use the XPath `node( )` test. This stylesheet:

```
<?xml version="1.0"?>
<!-- copy-identity.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="node()|@*">
    <xsl:copy>
      <xsl:apply-templates select="node()|@*"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

accomplishes the same thing as the `<xsl:copy-of>` stylesheet we looked at earlier.

For an example that needs that contol, we'll look a stylesheet that copies only the `<author>` information and the title of the sonnet. This stylesheet uses a separate template to make sure nothing happens to the `<lines>` element that contains the poem:

```
<?xml version="1.0"?>
<!-- copy4.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:apply-templates select="*"/>
  </xsl:template>

  <xsl:template match="*">
    <xsl:copy>
      <xsl:for-each select="@*">
        <xsl:copy/>
      </xsl:for-each>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="lines"/>
```

```
</xsl:stylesheet>
```

Our template for the <lines> element is empty; because it doesn't contain anything, it doesn't generate any output. That means our output should contain everything except the <lines> element and its children. The results look like this:

```
<?xml version="1.0" encoding="UTF-8"?><sonnet type="Shakespearean">
  <auth:author xmlns:auth="http://www.authors.com/">
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
    <year-of-death>1616</year-of-death>
  </auth:author>

  <title>Sonnet 130</title>

</sonnet>
```

The results contain all of the <auth:author> and <title> information, along with some extra whitespace. (Feel free to change the stylesheet to delete that whitespace if you want.) We couldn't do this with <xsl:copy-of>; <xsl:copy> gives us the control to do what we want. If we added more elements to our <sonnet> schema, this stylesheet would still copy everything except the <lines> elements and its children.

Our last stylesheet used an empty <xsl:template> for the <lines> element. That means all other elements, text, and attributes are copied to the output; our stylesheet defines what should *not* be copied to the output. Creating a stylesheet that defines what *should* be copied is more complex:

```
<?xml version="1.0"?>
<!-- copy5.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:auth="http://www.authors.com/">

  <xsl:template match="sonnet">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates select="auth:author|title"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="*">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

There are a couple of complications here. First of all, to specify that we want to copy the namespace-qualified element <auth:author>, we have to define that namespace in the stylesheet. Next, we list the two child elements of the <sonnet> element. Finally, we create a generic template that copies whatever elements are processed. Using the built-in template rules, the <auth:author> and <title> elements *and all their descendants* are processed with the generic template. We never process the <lines> element, so we get the results we want:

```
<?xml version="1.0" encoding="UTF-8"?><sonnet type="Shakespearean"><auth:author
xmlns:auth="http://www.authors.com/">
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
    <year-of-death>1616</year-of-death>
  </auth:author><title>Sonnet 130</title></sonnet>
```

We've looked at the two elements XSLT uses for copying: <xsl:copy-of> and <xsl:copy>. Use <xsl:copy-of> when you want a deep copy of an element, including its children and its attributes. On the other hand, <xsl:copy> forces you to choose exactly what gets copied. If you need that control, use <xsl:copy>; otherwise, use <xsl:copy-of>.

Dealing with Whitespace

One of the challenges of working with any XML document is processing whitespace, especially when want want to generate output other than HTML. As we noted earlier, an HTML browser renders both of these paragraphs the same way:

```
<p>
   This document contains
   5
   chapters.

</p>
<p>This document contains 5 chapters.</p>
```

If we generated these two paragraphs as text, however, they would appear as differently in print as they do in the HTML source. We'll look at several techniques for controlling whitespace in this section.

Whitespace Basics

Before we begin, it's worth defining the four characters that the XML spec defines as whitespace:

- The *tab* character (	)
- The *newline* character (
)

- The *carriage return* character (`
`)
- The *space* character (` `)

We'll use this modified version of our car list to illustrate how XML parsers and XSLT processors work with whitespace:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- carlist_whitespace.xml -->
<cars>
  <manufacturer name="          Chevrolet

">
    <car>Cavalier</car>
    <car>Corvette</car>
    <car>Impala</car>
    <car>Monte



Carlo</car>
  </manufacturer>
</cars>
```

From an XML parser's perspective, there are a number of whitespace-only nodes (nodes that contain only whitespace characters) in this document. The `<cars>` element contains a whitespace-only node with the newline character and the tab or spaces before the `<manufacturer>` tag, the node for the `<manufacturer>` element, and a whitespace-only node with the newline character after the `</manufacturer>` tag. Similarly, the `<manufacturer>` element contains whitespace-only nodes between the various `<car>` elements.

The XML parser doesn't remove any whitespace-only nodes, so we can always use them in our stylesheets. Put another way, the data model used by the XSLT processor contains the whitespace-only nodes from the XML source. The one exception to this is in an XSLT 2.0 processor that validates the XML source against a schema. If the schema indicates that an element can only contain other elements, any whitespace-only nodes contained in those elements are removed.

An XSLT processor doesn't know if the XML parser validated the document or not, so it assumes any whitespace nodes delievered by the XML parser must be significant. We'll use our sample stylesheet for the `<xsl:copy-of>` element to see what we get from our document. Processing our short list of cars with this stylesheet:

```
<?xml version="1.0"?>
<!-- copy-of.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:copy-of select="."/>
  </xsl:template>
</xsl:stylesheet>
```

gives us this XML document:

```
<?xml version="1.0" encoding="UTF-8"?><!-- carlist_whitespace.xml --><cars>
  <manufacturer name="          Chevrolet  ">
    <car>Cavalier</car>
    <car>Corvette</car>
    <car>Impala</car>
    <car>Monte



Carlo</car>
  </manufacturer>
</cars>
```

There's only one change in the processed version of the document: The newline characters were replaced with spaces in the value of the `name` attribute of the `<manufacturer>` element. All the whitespace nodes were preserved.

Using <xsl:preserve-space> and <xsl:strip-space>

The XSLT spec gives us two ways of dealing with whitespace nodes. We can use the `<xsl:preserve-space>` and `<xsl:strip-space>` nodes to keep or delete whitespace. Here's a slightly modified version of our stylesheet:

```
<?xml version="1.0"?>
<!-- copy-of-whitespace.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:preserve-space elements="manufacturer"/>
  <xsl:strip-space elements="cars"/>

  <xsl:template match="/">
    <xsl:copy-of select="."/>
  </xsl:template>
</xsl:stylesheet>
```

Here's what we get when we process our list of cars with the modified stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?><!-- carlist_whitespace.xml --><cars>
        <manufacturer name="          Chevrolet  ">
    <car>Cavalier</car>
    <car>Corvette</car>
    <car>Impala</car>
    <car>Monte



Carlo</car>
  </manufacturer></cars>
```

In the result document, the whitespace nodes inside the `<cars>` element have been removed, while all the whitespace inside the `<manufacturer>` element has been

preserved. (The complete text of the <manufacturer> element scrolls off the right side of the page.) You can use <xsl:preserve-space> and <strip-space> with wildcards as well:

```
<xsl:preserve-space elements="cars manufacturer"/>
<xsl:strip-space elements="*"/>
```

This tells the XSLT processor to strip whitespace nodes on all the elements except the <cars> and <manufacturer> elements. The elements attribute can contain an asterisk or a space-separated list of element names.

Because whitespace-only nodes are always preserved, there's no need to use <xsl:preserve-space elements="*"/>. Using <xsl:strip-space elements="*"/> as we have here means whitespace-only nodes are removed by default. We have to specify the mixed content elements in which whitespace is significant, but removing as many whitespace-only nodes as possible can make the node tree much smaller for a large document with lots of whitespace.

You can also use wildcards for namespace-qualified elements. For example, elements="auth:*" means all elements in the auth namespace. *[2.0]* In XSLT 2.0, you can also use elements="*:car to specify all <car> elements, regardless of their namespace.

The <xsl:preserve-space> and <strip-space> elements give us control over which whitespace nodes will be processed by our stylesheet. If we want to copy that whitespace directly to the output document, we can use the <xsl:copy-of> and <xsl:copy> elements as we discussed earlier.

A final note: the XML spec defines the little-used xml:space attribute. If an element in the XML source document contains the attribute xml:space="preserve", all whitespace in that element is preserved, regardless of any <xsl:strip-space> elements our stylesheet might have.

The normalize-space( ) function

Another useful technique for controlling whitespace is the normalize-space( ) function. In our previous example, we used <xsl:preserve-space> and <xsl:strip-space> to control whitespace nodes in various elements, but we still have quite a bit of whitespace in the name attribute and the last <car> in the list. To clean up the whitespace, we can use the normalize-space( ) function. It does three things:

- It removes all leading spaces.
- It removes all trailing spaces.
- It replaces any group of consecutive whitespace characters with a single space.

We'll use normalize-space( ) in this stylesheet:

```
<?xml version="1.0"?>
<!-- normalize-space.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:apply-templates />
  </xsl:template>

  <xsl:template match="*">
    <xsl:copy>
      <xsl:for-each select="@*">
        <xsl:attribute name="{name()}">
          <xsl:value-of select="normalize-space()"/>
        </xsl:attribute>
      </xsl:for-each>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="text()">
    <xsl:value-of select="normalize-space()"/>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates these crowded results:

```
<?xml version="1.0" encoding="UTF-8"?><cars><manufacturer name="Chevr
olet"><car>Cavalier</car><car>Corvette</car><car>Impala</car><car>Mon
te Carlo</car></manufacturer></cars>
```

This removes all the extraneous whitespace from the `name` attribute and the `<car>` element; it also effectively removes the whitespace-only text nodes.

A Simple Technique for Adding Whitespace to Text Output

Whenever we generate text output, we usually need to control line breaks. Programming languages have facilities for this; in Java, for example, we can use `System.out.print( )`, `System.out.println( )`, or even `System.out.print("\n\n")` to put line breaks exactly where we need them.

The simplest way to do this in an XSLT stylesheet is to use the character entities for the newline (`
`) and tab (`	`) characters. We've used this technique with the newline character throughout this chapter. As a further example, here's a stylesheet that uses all three character entities in the `separator` attribute of the `<xsl:value-of>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">

  <xsl:output method="text"/>

  <xsl:template match="/">
```

```
    <xsl:text>&#xA;Values separated with newlines:&#xA;&#xA;</xsl:text>
    <xsl:value-of select="1 to 7" separator="&#xA;"/>

    <xsl:text>&#xA;&#xA;Values separated with tabs:&#xA;&#xA;</xsl:text>
    <xsl:value-of select="1 to 7" separator="&#x9;"/>

    <xsl:text>&#xA;&#xA;Values separated with spaces:&#xA;&#xA;</xsl:text>
    <xsl:value-of select="1 to 7" separator="&#x20;"/>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates these results:

```
Values separated with newlines:

1
2
3
4
5
6
7

Values separated with tabs:

1       2       3       4       5       6       7

Values separated with spaces:

1 2 3 4 5 6 7
```

Summary

This chapter has covered the various ways you can generate output from your XSLT stylesheets. You'll use those basic techniques in every stylesheet you write. The main challenge as we go forward is learning how to select and organize the elements you want to process. You might need logic to select elements that have certain properties or you might need to sort or group elements before you process them. You might need to use special functions not defined in XSLT or XPath. You might need to write output to more than one document. Future chapters will address all of those topics.

Whatever your stylesheet does, you'll ultimately use the output methods in this chapter.

CHAPTER 5

Branching and Control Elements

So far, we've done some straightforward transformations and we've been able to do some reasonably sophisticated things. To do truly useful work, though, we'll need to use logic in our stylesheets. In this chapter, we'll discuss the XSLT elements that allow you to do just that. Although you'll see several XML elements that look like constructs from other programming languages, they're not exactly the same. As we go along, we'll discuss what makes XSLT different and how to do common tasks with your stylesheets.

Goals of This Chapter

By the end of this chapter, you should:

- Know the XSLT elements used for branching and control
- Understand the differences between XSLT's branching elements and similar constructs in other programming languages
- Know how to invoke XSLT templates by name and how to pass parameters to them, if you want
- Know how to use XSLT variables
- Understand how changes in XSLT 2.0 affect the way parameters and variables work
- Understand how to use recursion to get around the "limitations" of XSLT's branching and control elements

Branching Elements of XSLT

Three XSLT elements are used for branching: `<xsl:if>`, `<xsl:choose>`, and `<xsl:for-each>`. The first two are much like the `if` and `case` statements you may be familiar with from other languages, but the `for-each` element is significantly different from the `for` or `do-while` structures in other languages. We'll discuss all of them here.

The <xsl:if> Element

The `<xsl:if>` element looks like this:

```
<xsl:if test="count(zone) > 2">
  <xsl:text>Applicable zones: </xsl:text>
  <xsl:apply-templates select="zone"/>
</xsl:if>
```

The `<xsl:if>` element, surprisingly enough, implements an `if` statement. The element has only one attribute: `test`. If the value of `test` evaluates to the boolean value `true`, then all elements inside the `<xsl:if>` are processed. If `test` evaluates to `false`, then the contents of the `<xsl:if>` element are ignored. (If you want to implement an if-then-else statement, see the section "The <xsl:choose> Element" later in this chapter.)

Notice that we used the character `>` in the value of the `test` attribute. If you need to use the less-than operator (`<`), you'll have to use the `<` entity. The same holds true for the less-than-or-equal operator (`<=`).

Converting to boolean values

The `<xsl:if>` element is pretty simple, but it's the first time we've had to deal with *boolean values*. These values will come up later, so we might as well discuss them here. Attributes such as the `test` attribute of the `<xsl:if>` element convert whatever their values happen to be into a boolean value. If that boolean value is `true`, the `<xsl:if>` element is processed. (The `<xsl:when>` element, which we'll discuss in the section "The <xsl:choose> Element" later in this chapter, has a `test` attribute as well.)

[1.0] Here's the rundown of how various datatypes are converted to boolean values:

number
: If a number is positive or negative zero, it is `false`. If a numeric value is `NaN` (not a number; if I try to use the string "blue" as a number, the result is `NaN`), it is `false`. If a number has any other value, it is `true`.

node-set
: An empty node-set is `false`, a nonempty node-set is `true`.

string
: A zero-length string is `false`; a string whose length is not zero is `true`.

These rules are defined in Section 4.3 of the XPath 1.0 specification.

[2.0] For XSLT 2.0, things are more complicated. The value returned, as you'd expect, is an `xs:boolean`. Here's how different datatypes are converted to `xs:boolean` values:

- A singleton of any numeric type is `false` if its value is zero or `NaN` (not a number); everything else is `true`.
- A singleton of type `xs:string`, `xs:anyURI`, `xs:untypedAtomic`, or any type derived from them is `true` if the argument has a length greater than zero.

- The value of a singleton of type `xs:boolean` (or of a type derived from `xs:boolean`) is simply used as is.
- A sequence whose first item is a node is `true`.
- An empty sequence is `false`.
- Converting any other datatype to `xs:boolean` causes the XSLT processor to raise an error. For example, an `xs:date` can't be converted to `true` or `false`.

Boolean examples

Here are some examples that illustrate how boolean values evaluate the `test` attribute:

`<xsl:if test="count(zone) >= 2">`
: This is a boolean expression because it uses the greater-than-or-equal boolean operator. If the `count( )` function returns a value greater than or equal to 2, the `test` attribute is `true`. Otherwise, the `test` attribute is `false`.

`<xsl:if test="$x">`
: The variable `$x` is evaluated and converted to a boolean value using the rules we just covered. The result, of course, depends on the value of `$x` and how it is converted to a boolean value.

`<xsl:if test="true( )">`
: The boolean function `true( )` always returns the boolean value `true`. Therefore, this `test` attribute is always `true`.

`<xsl:if test="true">`
: This example is a trick. This `test` attribute is `true` only if there is at least one `<true>` element that's a child of the context node. The XSLT processor interprets the value `true` as an XPath expression that specifies all `<true>` elements in the current context. The strings `true` and `false` don't have any special significance in XSLT.

`<xsl:if test="'true'">`
: This `test` attribute is always `true`. Notice that in this case we used single quotes inside double quotes to specify that this is a literal string, not an element name. This `test` attribute is always `true` because the string has a length greater than zero, *not* because its value happens to be the word "true."

`<xsl:if test="'false'">`
: Another trick example; this `test` attribute is always `true`. As before, we used single quotes inside double quotes to specify that this is a literal string. Because the string has a length greater than zero, the `test` attribute is always `true`. The value of the nonempty string, confusing as it is, doesn't matter.

`<xsl:if test="not(3)">`
: This `test` attribute is always `false`. The literal `3` evaluates to `true`, so its negation is `false`. On the other hand, the expressions `not(0)` and `not(-0)` are always `true`.

`<xsl:if test="false( )">`
: This `test` attribute is always `false`. The boolean function `false( )` always returns the boolean value `false`.

`<xsl:if test="section/section">`
: The XPath expression `section/section` returns a node-set. If the current context contains one or more `<section>` elements that contain a `<section>` element in turn, the `test` attribute is `true`. If no such elements exist in the current context, the `test` attribute is `false`.

The <xsl:choose> Element

The `<xsl:choose>` element is logically equivalent to an if-then-else statement, although it has the feel of a `case` or `switch` statement in other programming languages. An `<xsl:choose>` contains at least one `<xsl:when>` element (logically equivalent to an `<xsl:if>` element), with an optional `<xsl:otherwise>` element (logically equivalent to an `else` in other programming languages). The `test` attribute of each `<xsl:when>` element is evaluated until the XSLT processor finds one that evaluates to `true`. When that happens, the contents of that `<xsl:when>` element are evaluated. (Unlike a `case` or `switch` element, each `<xsl:when>` is a separate test.) If none of the `<xsl:when>` elements have a test that is `true`, the contents of the `<xsl:otherwise>` element (if there is one) are processed.

<xsl:choose> example

Here's a sample `<xsl:choose>` element that sets the background color of the table's rows. If the `bgcolor` attribute is coded on the `<table-row>` element, the value of that attribute is used as the color; otherwise, the sample uses the `position( )` function and the `mod` operator to cycle the colors between `black`, `green`, `red`, and `blue`:

```
<xsl:template match="table-row">
  <tr>
    <xsl:attribute name="bgcolor">
     <xsl:choose>
        <xsl:when test="@bgcolor">
          <xsl:value-of select="@bgcolor"/>
        </xsl:when>
        <xsl:when test="position() mod 4 = 0">
          <xsl:text>black</xsl:text>
        </xsl:when>
        <xsl:when test="position() mod 4 = 1">
          <xsl:text>green</xsl:text>
        </xsl:when>
        <xsl:when test="position() mod 4 = 2">
          <xsl:text>red</xsl:text>
        </xsl:when>
        <xsl:otherwise>
          <xsl:text>blue</xsl:text>
        </xsl:otherwise>
      </xsl:choose>
```

```
      </xsl:attribute>
      <xsl:apply-templates select="*"/>
    </tr>
  </xsl:template>
```

In this sample, we use `<xsl:choose>` to generate the value of the `bgcolor` attribute of the `<tr>` element. Our first test is to see whether the `bgcolor` attribute of the `<table-row>` element exists; if it does, we use that value for the background color and the `<xsl:otherwise>` and other `<xsl:when>` elements are ignored. (If the `bgcolor` attribute is coded, the XPath expression `@bgcolor` returns a node-set containing a single attribute node.)

The next three `<xsl:when>` elements check the position of the current `<table-row>` element. The use of the `mod` operator here is the most efficient way to cycle between the various options. Finally, we use an `<xsl:otherwise>` element to specify `blue` as the default case. If `position( ) mod 4 = 3`, the background color will be `blue`.

There are a couple of minor details to note. In this example, we could replace the `<xsl:otherwise>` element with `<xsl:when test="position( ) mod 4 = 3">`; that is logically equivalent to the example as coded previously. For obfuscation bonus points, we could code the second `<xsl:when>` element as `<xsl:when test="not(position( ) mod 4)">`. (Remember that the boolean negation of zero is `true`.)

The <xsl:for-each> Element

If you want to process all the nodes that match a certain criteria, you can use the `<xsl:for-each>` element. Be aware that this isn't a traditional `for` loop; you can't ask the XSLT processor to do something like this:

```
for i = 1 to 10 do
```

The `<xsl:for-each>` element lets you select a set of nodes, and then do something with each of them. Let me mention again that this is not the same as a traditional `for` loop. Another important point is that the current node changes with each iteration through the `<xsl:for-each>` element. We'll go through some examples to illustrate this.

You can use the XSLT 2.0 `to` operator to do something similar (`select="1 to 10"`). When you're working with XSLT, it's better to think of `<xsl:for-each>` as an iterator rather than a traditional `for` loop.

<xsl:for-each> example

Here's a sample that selects all `<section>` elements inside a `<tutorial>` element and then uses a second `<xsl:for-each>` element to select all the `<panel>` elements inside each `<section>` element:

```
<xsl:template match="tutorial">
  <xsl:for-each select="section">
```

```
    <h1>
      <xsl:text>Section </xsl:text>
      <xsl:value-of select="position()"/>
      <xsl:text>. </xsl:text>
      <xsl:value-of select="title"/>
    </h1>
    <ul>
      <xsl:for-each select="panel">
        <li>
          <xsl:value-of select="position()"/>
          <xsl:text>. </xsl:text>
          <xsl:value-of select="title"/>
        </li>
      </xsl:for-each>
    </ul>
  </xsl:for-each>
</xsl:template>
```

Given this XML document:

```
<tutorial>
  <section>
    <title>Gene Splicing for Young People</title>
    <panel>
      <title>Introduction</title>
      <!-- ... -->
    </panel>
    <panel>
      <title>Discovering the secrets of life and creation</title>
      <!-- ... -->
    </panel>
    <panel>
      <title>"I created him for good, but he's turned out evil!"</title>
      <!-- ... -->
    </panel>
    <panel>
      <title>When angry mobs storm your castle</title>
      <!-- ... -->
    </panel>
  </section>
</tutorial>
```

The previous template produces these results:

```
<h1>Section 1. Gene Splicing for Young People</h1>
<ul>
  <li>1. Introduction</li>
  <li>2. Discovering the secrets of life and creation</li>
  <li>3. "I created him for good, but he's turned out evil!"</li>
  <li>4. When angry mobs storm your castle</li>
</ul>
```

Each time a `select` attribute is processed, it is evaluated in terms of the current node. As the XSLT processor cycles through all the `<section>` and `<panel>` elements, each of them in turn becomes the current node. By using iteration, we've generated a table of contents with a very simple template.

Invoking Templates by Name

Up to this point, we've always used XSLT's `<xsl:apply-templates>` element to invoke other templates. You can think of this as a limited form of *polymorphism;* a single instruction is invoked a number of times, and the XSLT processor uses each node in the node-set to determine which `<xsl:template>` to invoke. Most of the time, this is what we want. However, sometimes we want to invoke a particular template. XSLT allows us to do this with the `<xsl:call-template>` element.

How It Works

To invoke a template by name, two things have to happen:

- The template you want to invoke has to have a `name`.
- You use the `<xsl:call-template>` element to invoke the named template.

Here's how to do this. Say we have a template named *createMasthead* that creates the masthead of a web page. Whenever we create an HTML page for our web site, we want to invoke the *createMasthead* template to create the masthead. Here's what our stylesheet would look like:

```
<xsl:template name="createMasthead">
  <!-- interesting stuff that generates the masthead goes here -->
</xsl:template>
...
<xsl:template match="/">
  <html>
    <head>
      <title><xsl:value-of select="title"/></title>
    </head>
    <body>
      <xsl:call-template name="createMasthead"/>
...
```

Named templates are extremely useful for defining commonly used markup. For example, say you're using an XSLT stylesheet to create web pages with a particular look and feel. You can write named templates that create the header, footer, navigation areas, or other items that define how your web page will look. Every time you need to create a web page, simply use `<xsl:call-template>` to invoke those templates and create the look and feel you want.

Even better, if you put those named templates in a separate stylesheet and import the stylesheet (with either `<xsl:import>` or `<xsl:include>`), you can create a set of stylesheets that generate the look and feel of the web site you want. If you decide to redesign your web site, redesign the stylesheets that define the common graphical and layout elements. Change those stylesheets, regenerate your web site, and voila! You will see an instantly updated web site.

Templates à la mode

The XSLT <xsl:template> element has a mode attribute that lets you process the same set of nodes several times. For example, we might want to process <h1> elements one way when we generate a table of contents, and another way when we process the document as a whole. We could use the mode attribute to define different templates for different purposes:

```
<xsl:template match="h1" mode="build-toc">
  <!-- Template to process the <h1> element for table of contents -->
</xsl:template>

<xsl:template match="h1" mode="process-text">
  <!-- Template to process the <h1> element along with the rest   -->
  <!-- of the document                                            -->
</xsl:template>
```

We can then start applying templates with the mode attribute:

```
<xsl:template match="/">
  <html>
    <body>
      <h1>Table of Contents</h1>
      <ul>
        <xsl:apply-templates select="h1" mode="build-toc"/>
      </ul>
      <xsl:apply-templates select="*" mode="process-text"/>
    </body>
  </html>
</xsl:template>
```

This style of coding makes maintenance much easier; if the table of contents isn't generated correctly, the templates with mode="build-toc" are the obvious place to start debugging. (We discuss the mode attribute in more detail in the section "New values for the mode attribute" later in this chapter.)

Parameters

The XSLT <xsl:param> and <xsl:with-param> elements allow you to pass parameters to a template. You can pass templates with either the <call-template> element or the <apply-templates> element; we'll discuss the details in this section.

Defining a Parameter in a Template

To define a parameter in a template, use the <xsl:param> element. Here's an example of a template that defines two parameters:

```
<xsl:template name="calcuateArea">
  <xsl:param name="width"/>
  <xsl:param name="height"/>
```

```
  <xsl:value-of select="$width * $height"/>
</xsl:template>
```

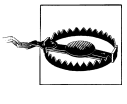
Conceptually, this is a lot like writing code in a traditional programming language, isn't it? Our template here defines two parameters, `width` and `height`, and outputs their product.

If you want, you can define a default value for a parameter. There are two ways to define a default value; the simplest is to use a `select` attribute on the `<xsl:param>` element:

```
<template name="addTableCell">
  <xsl:param name="bgColor" select="'blue'"/>
  <xsl:param name="width" select="150"/>
  <xsl:param name="content"/>
  <td width="{$width}" bgcolor="{$bgColor}">
    <xsl:apply-templates select="$content"/>
  </td>
</template>
```

In this example, the default values of the parameters `bgColor` and `width` are `'blue'` and `150`, respectively. If we invoke this template without specifying values for these parameters, the default values are used. Also notice that we generated the values of the `width` and `bgcolor` attributes of the HTML `<td>` tag with attribute value templates, the values in curly braces. For more information, see the section "Attribute Value Templates" in Chapter 3.

One thing to note about this example is that the `content` parameter doesn't have a default value here; we're assuming that `content` contains the nodes to be processed and put inside the table cell. If the value of `content` is an empty string, calling `<xsl:apply-templates select="$content"/>` causes an error. To be on the safe side, we could add an `<xsl:if>` element here to create an empty table cell if `content` is an empty string. As an exercise for the reader, feel free to make this code more robust....



Notice that in the previous example, we put single quotes around the value `blue`, but we didn't do it around the value `150`. Without the single quotes around `blue`, the XSLT processor assumes we want to select all the `<blue>` elements in the current context, which is probably not what we want. The XSLT processor is clever enough to realize that the value `150` can't be an XML element name (the XML 1.0 Specification says element names can't begin with numbers), so we don't need the single quotes around a numeric value.

Try to keep this in mind when you're using parameters. You'll probably forget it at some point, and you'll probably go nuts trying to figure out the strange behavior you're getting from the XSLT processor.

The second way to define a default value for a parameter is to include content inside the `<xsl:param>` element:

```
<template name="addTableCell">
  <xsl:param name="bgColor">
```

```
    <xsl:text>blue</xsl:text>
  </xsl:param>
  <xsl:param name="width">
    <xsl:value-of select="7+8"/><xsl:text>0</xsl:text>
  </xsl:param>
  <xsl:param name="content"/>
  <td width="{$width}" bgcolor="{$bgColor}">
    <xsl:apply-templates select="$content"/>
  </td>
</template>
```

In this example, we used `<xsl:text>` and `<xsl:value-of>` elements to define the default values of the parameters. Out of sheer perverseness, we defined the value of `width` as the concatenation of the numeric expression `7+8`, followed by the string "0". The result of the numeric expression, `15`, is converted to a string, and then that string is concatenated with the string `0` This example produces the string `150`, which will be converted into a number as necessary.

Passing Parameters

If we invoke a template by name, which is similar to calling a subroutine, we'll need to pass parameters to those templates. We do this with the `<xsl:with-param>` element. For example, let's say we want to call a template named *draw-box*, and then pass the parameters `startX`, `startY`, `endX`, and `endY` to it. Here's what we'd do:

```
<xsl:call-template name="draw-box">
  <xsl:with-param name="startX" select="50"/>
  <xsl:with-param name="startY" select="50"/>
  <xsl:with-param name="endX" select="97"/>
  <xsl:with-param name="endY" select="144"/>
</xsl:call-template>
```

In this sample, we've called the template named `draw-box` with the four parameters we mentioned earlier. Notice that up until now, `<xsl:call-template>` has always been an empty tag; here, though, the parameters are the content of the `<xsl:call-template>` element. (If you want, you can do the same thing with `<xsl:apply-templates>`.)

If we're going to pass parameters to a template, we have to set up the template so that it expects the parameters we're passing. To do this, we'll use the `<xsl:param>` element inside the template. Here are some examples:

```
<xsl:template name="draw-box">
  <xsl:param name="startX"/>
  <xsl:param name="startY" select="'0'"/>
  <xsl:param name="endX">
    10
  </xsl:param>
  <xsl:param name="endY">
    10
  </xsl:param>
  ...
</xsl:template>
```

A couple of notes about the <xsl:param> element:

- If you define any <xsl:param> elements in a template, they must be the first thing in the template.
- The <xsl:param> element allows you to define a default value for the parameter. If the calling template doesn't supply a value, the default is used instead. The last three <xsl:param> elements in our previous example define default values.
- The <xsl:param> element has the same content model as <xsl:variable>. With no content and no select attribute, the default value of the parameter is an empty string (""). With a select attribute, the default value of the parameter is the value of the select attribute. If the <xsl:param> element contains content, the default value of the parameter is the content of the <xsl:param> element.

Global Parameters

XSLT allows you to define parameters whose scope is the entire stylesheet. You can define default values for these parameters, and you can pass values to those parameters externally to the stylesheet. Before we talk about how to pass in values for *global parameters*, we'll show you how to create them. Any parameters that are top-level elements (any <xsl:param> elements whose parent is <xsl:stylesheet>) are global parameters. Here's an example:

```
<?xml version="1.0"?>
<!-- params.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:param name="startX"/>
  <xsl:param name="baseColor"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Global parameters example&#xA;&#xA;</xsl:text>

    <xsl:text>The value of startX is: </xsl:text>
    <xsl:value-of select="$startX"/>
    <xsl:text>&#xA;The value of baseColor is: </xsl:text>
    <xsl:value-of select="$baseColor"/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

How you pass values for global parameters depends on the XSLT processor you're using. We'll go through some examples here for all the usual suspects. Let's say we want to pass the numeric value 50 as the value for startX, and the string value magenta as the default value for baseColor. The following list describes the commands you'd use to do that:

Xalan

To pass global parameters to Xalan, you can define them on the Xalan command line:

```
java org.apache.xalan.xslt.Process -in blank.xml -xsl params.xsl
  -param startX 50 -param baseColor magenta
```

(This command should be on a single line.)

Saxon

Saxon supports external parameters like this:

```
java net.sf.saxon.Transform blank.xml params.xsl startX=50 baseColor=magenta
```

Microsoft's XSLT tools

Here's how you pass external parameters to Microsoft's XSLT tools:

```
msxsl blank.xml params.xsl startX=50 baseColor=magenta
```

Altova

If you're using the Altova XML engine, you pass external parameters like this:

```
altovaxml /xslt1 params.xsl /in blank.xml /param startX=50 /param
baseColor='magenta'
```

Notice that we have to put single quotes around the text value `magenta`.

Using this stylesheet with any XML document and any of the XSLT processors listed here produces these results:

```
Global parameters example

The value of startX is: 50
The value of baseColor is: magenta
```

Setting global parameters in a Java program

If your XSLT engine supports the Transformation API for XML (TrAX), you can embed the XSLT processor and set global parameters in your code. Here's an example that uses TrAX support:

```
import java.io.File;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

public class GlobalParameters
{
  public static void parseAndProcess(String sourceID,
                                     String xslID,
                                     String outputID)
  {
    try
```

```
    {
      TransformerFactory tfactory = TransformerFactory.newInstance();

      Transformer transformer
        = tfactory.newTransformer(new StreamSource(xslID));

      // Use the setParameter method to set global parameters
      transformer.setParameter("startX", new Integer(50));
      transformer.setParameter("baseColor", "magenta");

      transformer.transform(new StreamSource(sourceID),
                            new StreamResult(outputID));
    }
    catch (TransformerConfigurationException tce)
    {
      System.err.println("Exception: " + tce);
    }
    catch (TransformerException te)
    {
      System.err.println("Exception: " + te);
    }
  }

  public static void main(String argv[])
    throws java.io.IOException,
           org.xml.sax.SAXException
  {
    GlobalParameters gp = new GlobalParameters();
    gp.parseAndProcess(argv[0], argv[1], argv[2]);
  }
}
```

Notice that we used the `setParameter` method to set global parameters for the `Transformer` object before we invoke the `transform` method. This transformation generates the following results in *output.text*:

```
Global parameters example

The value of startX is: 50
The value of baseColor is: magenta
```

Setting global parameters in .NET

The .NET framework provides the `XsltArgumentList` object for setting global stylesheet parameters. As with the Java example, the code is straightforward:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Xml.Xsl;

namespace com.oreilly.xslt
{
  class XsltGlobalParameters
```

```
{
  static void Main(string[] args)
  {
    // Create the stylesheet object and the XMLWriter that
    // writes the output to a file
    XslCompiledTransform stylesheet = new XslCompiledTransform();
    XmlTextWriter xWriter =
      new XmlTextWriter(args[2], Encoding.UTF8);

    // Use an XsltSettings object that allows executing scripts
    // (we need this for extensions), then load the stylesheet
    XsltSettings settings = new XsltSettings(true, true);
    stylesheet.Load(args[1], settings, new XmlUrlResolver());

    // We pass global parameters to the stylesheet with an
    // XsltArgumentList object.
    XsltArgumentList argList = new XsltArgumentList();
    argList.AddParam("startX", "", 50);
    argList.AddParam("baseColor", "", "magenta");

    // With everything in place, we call the Transform() method
    // to do the work...
    stylesheet.Transform(args[0], argList, xWriter);
  }
 }
}
```

We create an `XslCompiledTransform` object, and then use an `XsltArgumentList` object to set parameters for the stylesheet. Once everything is set, we use the `Transform( )` method of the `XslCompiledTransform` class. This generates the same results we saw with the Java code:

```
Global parameters example

The value of startX is: 50
The value of baseColor is: magenta
```

[2.0] Important Differences in XSLT 2.0

There are some changes to the way parameters and modes work in XSLT 2.0. We'll cover those here. To summarize, the key differences are:

- The `mode` attribute features three new values: `#all`, `#current`, and `#default`.
- If you pass a parameter to a template, and that parameter is not defined in that template, an XSLT 2.0 processor will give you an error message and stop. In XSLT 1.0, the undefined parameters were simply ignored.
- XSLT 2.0 adds the attribute `required="yes"` to define that a value must be passed for a parameter.
- You can specify the datatype and/or the structure of a parameter. If a parameter must be an `xs:date` or a sequence of at least one element, you can specify that.

- XSLT 2.0 defines a new kind of parameter called a *tunnel parameter*. Tunnel parameters help you avoid sloppy coding practices that you were often forced into with XSLT 1.0.

New values for the mode attribute

In XSLT 2.0, there are three new values for the `mode` attribute:

`#all`
: For `<xsl:template>`, we can use the value `mode="#all"`. This specifies that a given template matches all modes. However, if the current mode is `"toc"`, a template with `mode="toc"` is invoked instead of a template with `mode="#all"`.

`#current`
: For the `<xsl:apply-templates>` element, we can use the value `mode="#current"` to invoke other templates using the current mode. This effectively uses the current mode as a parameter.

`#default`
: The `<xsl:apply-templates>` and `<xsl:template>` elements can use `mode="#default"`. The default mode is unnamed.

Another difference in XSLT 2.0: the value of the `mode` attribute can be a space-separated list of mode names. In XSLT 1.0, you could only specify one mode at a time.

See the definition of the `<xsl:apply-templates>` element in Appendix A for a complete example of these new features.

Undefined parameters are illegal

In XSLT 1.0, you can pass as many parameters to a template as you want. If you pass two parameters to a template that defines only one parameter, the extra parameter is simply ignored. You won't even get a warning or error message from the XSLT processor. Here's an example:

```
<?xml version="1.0"?>
<!-- parameters-1_0.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:call-template name="test">
      <xsl:with-param name="param1" select="'57'"/>
      <xsl:with-param name="param2" select="'93'"/>
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="test">
    <xsl:param name="param1"/>
    <xsl:text>Value of $param1: </xsl:text>
```

```
    <xsl:value-of select="$param1"/>
  </xsl:template>

</xsl:stylesheet>
```

In this example, the template for the document root calls a named template, passing it two parameters. The parameter `param2` isn't defined in the named template. With an XSLT 1.0 processor, this is *not* an error. Here's the output you get from Xalan-J 2.7.0, which is an XSLT 1.0 processor:

```
java org.apache.xalan.xslt.Process -xsl parameters-1_0.xsl
Value of $param1: 57
```

Using the same stylesheet with Saxon 9.0.0.3, which is an XSLT 2.0 processor, gives these results:

```
java net.sf.saxon.Transform blank.xml parameters-1_0.xsl
Warning: Running an XSLT 1.0 stylesheet with an XSLT 2.0 processor
Value of $param1: 57
```

Saxon gives us a warning message that we're running an XSLT 1.0 stylesheet, but it ignores the extra parameter as it should when it's processing a stylesheet in XSLT 1.0 mode. However, if we change the `version` attribute of the `<xsl:stylesheet>` tag to look like this:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">
```

Saxon processes this as an XSLT 2.0 stylesheet:

```
java net.sf.saxon.Transform blank.xml parameters-2_0.xsl
Error at xsl:call-template on line 9 of file:/C:/parameters-2_0.xsl:
  XTSE0680: Parameter param2 is not declared in the called template
Failed to compile stylesheet. 1 error detected.
```

An XSLT 2.0 stylesheet processor stops the transformation as soon as it finds the undefined parameter.

Finally, if we run the XSLT 2.0 stylesheet with Xalan, it ignores the `version="2.0"` attribute entirely and processes the stylesheet as you'd expect an XSLT 1.0 processor to do:

```
java org.apache.xalan.xslt.Process -xsl parameters-2_0.xsl
Value of $param1: 57
```

You might have noticed in the preceding code that when we processed the stylesheet with Xalan, we specified only our stylesheet. Saxon, on the other hand, requires that we specify both an XML file and an XSLT stylesheet. The contents of the file *blank.xml* are the single empty element `<blank/>`.

An alternative would be to give the template a name (`<xsl:template match="/" name="main">`) and invoke it using Saxon's `-it` option:

```
java net.sf.saxon.Transform -it main parameters_1.0.xsl
Warning: Running an XSLT 1.0 stylesheet with an XSLT 2.0 processor
Value of $param1: 57
```

The `-it` option lets you specify the named template where the transformation should begin.

Required parameters

XSLT 2.0 adds a `required` attribute to the `<xsl:param>` element. Valid values are `yes` and `no`, as you'd expect. If a parameter is required, the `<xsl:param>` element *must not* have a `select` attribute.

Here's an example of a required parameter:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- required_parameters.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:call-template name="date-formatter">
      <xsl:with-param name="date" select="current-date()"/>
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="date-formatter">
    <xsl:param name="date" required="yes"/>
    <xsl:value-of select="format-date($date, '[M01]/[D01]/[Y0001]')"/>
  </xsl:template>

</xsl:stylesheet>
```

In this stylesheet, the `date` parameter is required; if we try to invoke this template without passing the required parameter, the XSLT processor will refuse to run our stylesheet. This would be a good place to use XSLT 2.0's datatyping support (more on this next); the XSLT processor forces us to use the required parameter, but it doesn't check its datatype. If the parameter we pass to the `format-date( )` function is not an `xs:date`, the XSLT processor throws an error. (This stylesheet uses the `current-date( )` function to generate the parameter, so we know we'll always have the correct datatype.)

A final restriction on <xsl:param> is that you can't use the required and select attributes on the same parameter. Remember, the select attribute defines a default value in case a parameter isn't passed to the template. On the other hand, the required attribute says that the parameter *must* be passed to the template.

Datatyping support

The <xsl:param>, <xsl:with-param>, and <xsl:variable> elements have an optional as attribute that define the datatype and/or structure of a parameter or variable. As an example, here's a parameter that must be an xs:date:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- datatype_parameters.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:call-template name="date-formatter">
      <xsl:with-param name="date" select="current-date()"/>
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="date-formatter">
    <xsl:param name="date" as="xs:date" required="yes"/>
    <xsl:value-of select="format-date($date, '[M01]/[D01]/[Y0001]')"/>
  </xsl:template>

</xsl:stylesheet>
```

Typically we're concerned about the datatype of a parameter, but we can also define the parameter's structure. The attribute as="element( )+" says the parameter must be a sequence of one or more element nodes, while as="element(*, xs:date)+" says that the parameter must be a sequence of one or more element nodes, each of which has the datatype xs:date.

In this stylesheet, we're calling a named template. The <xsl:param> element in the named template tells the XSLT processor that the parameter is required, and that its datatype must be the XML Schema date type. When we run this stylesheet with an XSLT 2.0 processor, we get these results:

```
java net.sf.saxon.Transform blank.xml datatype_parameters.xsl
03/02/2008
```

The XSLT 2.0 stylesheet engine has taken the value of our parameter and formatted it according to the picture clause used in the format-date( ) function.

If the parameter's datatype doesn't match the datatype required by the template, an error will occur. For example, if we change the stylesheet so that we call the template with bad data:

```
<xsl:call-template name="date-formatter">
  <xsl:with-param name="date" select="'blue'"/>
</xsl:call-template>
```

we'll get an error:

```
Error at xsl:with-param on line 11 of file:/C:/datatype_parameters.xsl:
  XTTE0570: Required item type of value of variable $date is xs:date;
  supplied value has item type xs:string
Failed to compile stylesheet. 1 error detected.
```

The stylesheet engine doesn't even process the entire stylesheet because the parameter's value (the string `blue`) clearly doesn't match the required datatype. If we generated the value of the parameter dynamically and the generated value wasn't an `xs:date`, we would get a runtime error that would stop the transformation.

Tunnel parameters

The use of parameters in XSLT 1.0 can lead to sloppy programming in a couple of ways. First of all, you don't have to worry so much about the "signature" of the template you're invoking. You can pass the exact number of parameters to the template, none at all, or twice as many. The XSLT 1.0 processor will probably do what you want, but if a parameter isn't set correctly, it can be difficult to figure out where the problem lies. XSLT 2.0's requirement that you pass the exact number of parameters makes for much cleaner code.

The second problem is when you need to pass a parameter that might eventually be used by another template. As an example, say we have a stylesheet that generates HTML from DocBook. DocBook features hundreds of elements, so we'll just look at templates for a few DocBook elements here. We'll take a DocBook document and create two HTML files. Each HTML file will contain the major section headings (`sect1/title`), along with all of the code listings in the source document (DocBook `<programlisting>` elements). We'll run this transformation twice, generating normal-sized text in one document and larger text in the second.

Here's what our stylesheet looks like *without* tunnel parameters:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- normal_parameters.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <xsl:result-document href="regular-type.html" method="html">
      <html>
        <xsl:apply-templates select="*|text()">
```

```
        <xsl:with-param name="code-font-size" select="'14'"/>
      </xsl:apply-templates>
    </html>
  </xsl:result-document>
  <xsl:result-document href="larger-type.html" method="html">
    <html>
      <xsl:apply-templates select="*|text()">
        <xsl:with-param name="code-font-size" select="'20'"/>
      </xsl:apply-templates>
    </html>
  </xsl:result-document>
</xsl:template>

<xsl:template match="chapter">
  <xsl:param name="code-font-size"/>
  <head>
    <title><xsl:value-of select="title"/></title>
  </head>
  <body>
    <xsl:apply-templates select="*[not(name() = 'title')]|text()">
      <xsl:with-param name="code-font-size" select="$code-font-size"/>
    </xsl:apply-templates>
  </body>
</xsl:template>

<xsl:template match="programlisting">
  <xsl:param name="code-font-size"/>
  <pre>
    <span>
      <xsl:attribute name="style">
        <xsl:text>font-family:monospace; font-size:</xsl:text>
        <xsl:value-of select="$code-font-size"/>
        <xsl:text>;</xsl:text>
      </xsl:attribute>
      <xsl:apply-templates select="*|text()">
        <xsl:with-param name="code-font-size" select="$code-font-size"/>>
      </xsl:apply-templates>
    </span>
  </pre>
</xsl:template>

<xsl:template match="sect1/title">
  <xsl:param name="code-font-size"/>
  <h1>
    <xsl:apply-templates select="*|text()">
      <xsl:with-param name="code-font-size" select="$code-font-size"/>
    </xsl:apply-templates>
  </h1>
</xsl:template>

<!-- A useful stylesheet would have dozens more templates here... -->
<xsl:template match="*">
  <xsl:param name="code-font-size"/>
  <xsl:apply-templates select="*">
    <xsl:with-param name="code-font-size" select="$code-font-size"/>
```

```
    </xsl:apply-templates>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates the output we want, creating the result documents `regular-type.html` and `larger-type.html`. An excerpt from `regular-type.html` looks like this:

```
      <h1>Goals of This Chapter</h1>

      <h1>Branching Elements of XSLT</h1><pre>
<span style="font-family:monospace; font-size:14;">
&lt;xsl:if test="count(zone) &gt; 2"&gt;
  &lt;xsl:text&gt;Applicable zones: &lt;/xsl:text&gt;
  &lt;xsl:apply-templates select="zone"/&gt;
&lt;/xsl:if&gt;</span></pre><pre>
<span style="font-family:monospace; font-size:14;">
&lt;xsl:template match="table-row"&gt;
  &lt;tr&gt;
    &lt;xsl:attribute name="bgcolor"&gt;
     &lt;xsl:choose&gt;
```

The file `larger-type.html` is identical, with the exception that the `style` attribute contains `font-size:20;` instead of `font-size:14;`. However, the stylesheet is messy, and maintenance will be more difficult than it should be. Notice how many templates have this structure:

```
<xsl:template match="whatever">
  <xsl:param name="code-font-size"/>

  <!-- Do something with the current element, -->
  <!-- then process its descendants           -->

  <xsl:apply-templates select="*|text()">
    <!-- Pass the code-font-size parameter, just in case -->
    <!-- we need it later -->
    <xsl:with-param name="code-font-size" select="$code-font-size"/>
  </xsl:apply-templates
</xsl:template>
```

The problem is that everytime we use `<xsl:apply-templates>`, we have to pass along the `code-font-size` variable just in case a template somewhere down the line needs it. Any of the elements for which we've written templates might have `<programlisting>` as a descendant, so we don't have any choice. To make things even worse, every time we add a new template to our stylesheet, we have to add the same `<xsl:param>` and `<xsl:with-param>` markup. If we suddenly had three more parameters that we needed to pass around in this way, our stylesheet would become very convoluted, and every change to the stylesheet could introduce errors if we don't remember to make the same changes to all the affected templates.

And that's where tunnel parameters come in.

Tunnel parameters are similar to *dynamically scoped variables* in functional programming languages such as Haskell and Scheme. In those languages, a variable may go in and out of scope as one function invokes another. In XSLT 2.0, when you create a tunnel parameter, that parameter is passed on to each template that's directly or indirectly invoked. As one template invokes another during processing, any template anywhere can use that tunnel parameter simply by referring to it as a tunnel parameter. Here's how the stylesheet looks with tunnel parameters:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- tunnel_parameters.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <xsl:result-document href="regular-type.html" method="html">
      <html>
        <xsl:apply-templates select="*|text()">
          <xsl:with-param name="code-font-size" select="'14'"
            tunnel="yes"/>
        </xsl:apply-templates>
      </html>
    </xsl:result-document>
    <xsl:result-document href="larger-type.html" method="html">
      <html>
        <xsl:apply-templates select="*|text()">
          <xsl:with-param name="code-font-size" select="'20'"
            tunnel="yes"/>
        </xsl:apply-templates>
      </html>
    </xsl:result-document>
  </xsl:template>

  <xsl:template match="chapter">
    <head>
      <title><xsl:value-of select="title"/></title>
    </head>
    <body>
      <xsl:apply-templates select="*[not(name() = 'title')]|text()"/>
    </body>
  </xsl:template>

  <xsl:template match="programlisting">
    <xsl:param name="code-font-size" tunnel="yes"/>
    <pre>
      <span>
        <xsl:attribute name="style">
          <xsl:text>font-family:monospace; font-size:</xsl:text>
          <xsl:value-of select="$code-font-size"/>
          <xsl:text>;</xsl:text>
        </xsl:attribute>
        <xsl:apply-templates select="*|text()"/>
      </span>
```

```
    </pre>
  </xsl:template>

  <xsl:template match="sect1/title">
    <h1>
      <xsl:apply-templates select="*|text()"/>
    </h1>
  </xsl:template>

  <!-- A useful stylesheet would have dozens more templates here... -->
  <xsl:template match="*">
    <xsl:apply-templates select="*"/>
  </xsl:template>

</xsl:stylesheet>
```

Notice how tunnel parameters have simplified the code. In the root template, we're passing a tunnel parameter as we tell the XSLT processor to transform all of the descendant elements. Each time a subsequent template invokes another template, whether through `<apply-templates>` or `<call-template>`, the tunnel parameters are silently passed along. The only time we use the parameter is in the only place we need it: the template for the `programlisting` element.

A couple of syntax notes: first of all, the `<xsl:with-param>` element that declares the parameter must have `tunnel="yes"` to be a tunnel parameter. Secondly, the template that wants to use the tunnel parameter must have the `tunnel="yes"` attribute on the `<xsl:param>` element that defines the parameter. If the parameter definition doesn't include `tunnel="yes"`, the XSLT processor assumes that the parameter is a new variable local to that template.

It might have occurred to you that we could solve this problem with a global parameter. That's true, although there are a couple of disadvantages to this method. First of all, we would like to limit the number of global parameters. Adding a global parameter just so we can use it anywhere we need it isn't good form.

The second problem is that we can't change the value of the global parameter. In our example here, we would have to create two global variables, one for each value we'd like to use in the stylesheet. Using tunnel parameters lets us set the value of `code-font-size` each time we want to process our source document. With tunnel parameters, our stylesheet is easier to write, easier to debug, and easier to maintain.

Variables

If we use logic to control the flow of our stylesheets, we'll probably want to store temporary results along the way. In other words, we'll need to use variables. XSLT provides the `<xsl:variable>` element, which allows you to store a value and associate it with a name.

The `<xsl:variable>` element can be used in three ways. The simplest form of the element creates a new variable whose value is an empty string (`""`). Here's how it looks:

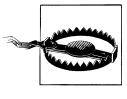
```
<xsl:variable name="x"/>
```

This element creates a new variable named `x`, whose value is an empty string. (Please hold your applause until the end of the section.)

You can also create a variable by adding a `select` attribute to the `<xsl:variable>` element:

```
<xsl:variable name="favouriteColour" select="'blue'"/>
```

In this case, we've set the value of the variable to be the string "blue". Notice that we put single quotes around the value. These quotes ensure that the literal value `blue` is used as the value of the variable. If we had left out the single quotes, this would mean the value of the variable is the node-set (or sequence) of all the `<blue>` elements in the context node, which definitely isn't what we want here.



Be aware that single quotes around numeric values are significant. The value `35` represents a numeric value (it's a number in XSLT 1.0, and an `xs:integer` in XSLT 2.0), while the value `'35'` represents the *string* `35`. That might seem like a minor distinction, but it has a major impact on how your stylesheet works, especially in XSLT 2.0.

The third way to use the `<xsl:variable>` element is to put content inside it. Here's a brief example:

```
<xsl:variable name="y">
  <xsl:choose>
    <xsl:when test="$x &gt; 7">
      <xsl:text>13</xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:text>15</xsl:text>
    </xsl:otherwise>
  </xsl:choose>
</xsl:variable>
```

In this more complicated example, the content of the variable `y` depends on the test attribute of the `<xsl:when>` element. This is the equivalent of this procedural programming construct:

```
int y;
if (x > 7)
  y = 13;
else
  y = 15;
```

Are These Things Really Variables?

Although these XSLT variables are called variables, they're not variables in the traditional sense of procedural programming languages such as C++ or Java. Remember that earlier we said one goal behind the design of the stylesheet language is to avoid side effects in execution? Well, one of the most common side effects used in most procedural languages is changing the value of a variable. If we write our stylesheet so that the results depend on the varying values of different variables, the stylesheet engine would be forced to evaluate the templates in a certain order.

XSLT variables are more like variables in the traditional mathematical sense. In mathematics, we can define a function called `square(x)` that returns the value of a number (represented by `x`) multiplied by itself. In other words, `square(2.5)` returns `6.25`. In this context, we understand that `x` can be any number; we also understand that the `square` function can't change the value of `x`.

It takes a while to get used to this concept, but you'll get there. Trust me on this.

Variable Scope

An `<xsl:variable>` element is scoped to the element that contains it. If an `<xsl:variable>` element is a top-level element (its parent is `<xsl:stylesheet>`), it is global, and its value is visible everywhere in the stylesheet. You can also use an `<xsl:variable>` element within an `<xsl:template>` to override the value of a global variable locally.

Using Recursion to Do Most Anything

Writing an XSLT stylesheet is different from programming in other languages. If you didn't believe that before, you probably do now. We'll finish this chapter with a couple of examples that demonstrate how to use recursion to solve the kinds of problems that you're probably used to solving with procedural programming languages. We'll also look at some new features of XSLT 2.0 and XPath 2.0 that allow you to avoid recursion in some situations.

Implementing a String Replace Function

To demonstrate how to use recursion to solve problems, we'll write a string replace function. This is sometimes useful when you need to escape certain characters or substrings in your output. The stylesheet we'll develop here transforms an XML document into a set of SQL statements that will be executed at a Windows command prompt. We have to do several things:

Put a caret (^) in front of all ampersands (&)
: On the Windows NT and Windows 2000 command prompt, the ampersand means that the current command has ended and another is beginning. For example, this command creates a new directory called *xslt* and changes the current directory to the newly created one:

```
mkdir xslt & chdir xslt
```

If we create a SQL statement that contains an ampersand, we'll need to escape the ampersand so it's processed as a literal character, not as an operator. If we insert the value `Jones & Son` as the value of the company field in a row of the database, we need to change it to `Jones ^& Son` before we try to run the SQL command.

Put a caret (^) in front of all vertical bars (|)
: The vertical bar is the pipe operator on Windows systems, so we need to escape it if we want it interpreted as literal text instead of an operator.

Replace any single quote (') with two single quotes ('')
: This is a requirement of our database system.

Procedural design

Three functions we could use in our template are `concat( )`, `substring-before( )`, and `substring-after( )`. To replace an ampersand with a caret and an ampersand, this would do the trick:

```
<xsl:value-of select="concat(substring-before(., '&amp;'), '^&amp;',
                             substring-after(., '&amp;'))"/>
```

The obvious problem with this step is that it replaces only the first occurrence of the ampersand. If there are two ampersands, or three, or three hundred, we need to call this method once for each ampersand in the original string. Because of the way variables work, we can't do what we'd do in a procedural language:

```
private static String strChange(String string, String from, String to)
{
  String before = "", after = "";
  int    index;

  index = string.indexOf(from);
  while (index >= 0)
  {
    before = string.substring(0, index);
    after = string.substring(index + from.length());
    string = before + to + after;

    index = string.indexOf(from, index + to.length());
  }

  return string;
}
```

XSLT doesn't have any simple way to iterate through the characters of the string, so we'll use recursion instead.

Recursive design

To implement a string replace function with recursion, we'll take this approach:

- If the whole string *does contain* the substring we want to replace, we do the following:
  1. Return the first part of the whole string—everything before the substring we want to replace.
  2. Return the replacement substring.
  3. Return the result of calling our function with the last part of the whole string—everything after the first occurrence of the substring we want to replace. *This is the recursive part of our design.*
- If the whole string *does not contain* the substring we want to replace, we simply return the whole string.

If the substring we're replacing occurs in the whole string, we call the substring replace function on the last of the string. The key here, as with all recursive functions, is that we have an *exit case*, a condition in which we don't recurse. Eventually we'll call our recursive function with a string that doesn't contain the substring we're replacing.

Here's the design in pseudocode:

```
replaceSubstring(originalString, substring, replacementString)
{
  if (contains(originalString, substring))
  {
    return
      (substring-before(originalString, substring) +
       replacementString +
       replaceSubstring(substring-after(originalString, substring),
                        substring, replacementString));
  }
  else
    return originalString;
}
```

In the recursive approach, the function calls itself whenever there's at least one occurrence of the substring. Each time the function calls itself, the `originalString` parameter is a little smaller, until eventually we've processed the complete string. Here's the complete stylesheet:

```
<?xml version="1.0"?>
<!-- string_replace-1_0.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>
```

```
<xsl:template match="/">
  <xsl:apply-templates select="ul/li"/>
</xsl:template>

<xsl:template match="li">
  <xsl:variable name="single-quote">
    <xsl:text>&apos;</xsl:text>
  </xsl:variable>
  <xsl:variable name="two-quotes">
    <xsl:text>&apos;&apos;</xsl:text>
  </xsl:variable>

  <xsl:variable name="sub1">
    <xsl:call-template name="replace-substring">
      <xsl:with-param name="original" select="."/>
      <xsl:with-param name="substring" select="'&amp;'"/>
      <xsl:with-param name="replacement" select="'^&amp;'"/>
    </xsl:call-template>
  </xsl:variable>

  <xsl:variable name="sub2">
    <xsl:call-template name="replace-substring">
      <xsl:with-param name="original" select="$sub1"/>
      <xsl:with-param name="substring" select="'|'"/>
      <xsl:with-param name="replacement" select="'^|'"/>
    </xsl:call-template>
  </xsl:variable>

  <xsl:call-template name="replace-substring">
    <xsl:with-param name="original" select="$sub2"/>
    <xsl:with-param name="substring" select="$single-quote"/>
    <xsl:with-param name="replacement" select="$two-quotes"/>
  </xsl:call-template>
  <xsl:text>&#xA;</xsl:text>
</xsl:template>

<xsl:template name="replace-substring">
  <xsl:param name="original" />
  <xsl:param name="substring" />
  <xsl:param name="replacement" />
  <xsl:choose>
    <xsl:when test="contains($original, $substring)">
      <xsl:value-of
        select="substring-before($original, $substring)" />
      <xsl:value-of select="$replacement" />
      <xsl:call-template name="replace-substring">
        <xsl:with-param name="original"
          select="substring-after($original, $substring)" />
        <xsl:with-param
          name="substring" select="$substring" />
        <xsl:with-param
          name="replacement" select="$replacement" />
      </xsl:call-template>
    </xsl:when>
```

```
      <xsl:otherwise>
        <xsl:value-of select="$original" />
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

</xsl:stylesheet>
```

We create the variable `$sub1` by replacing all of the ampersands in the original text with a caret and an ampersand. We create the variable `$sub2` by replacing all of the vertical bars in `$sub1` with a caret and a vertical bar. Finally we use `$sub2` in our third call to the `replace-substring` template. The third call to the template doubles all the single quotes. Notice that the third call isn't inside an `<xsl:variable>` element, so it is written to the output.

Given this XML input document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- testlines.xml -->
<ul>
  <li>This is a test &amp;amp; I hope it works | fails gracefully</li>
  <li>Some techniques are simpler &amp;amp; easier than recursion</li>
  <li>Will I enjoy next Tuesday's meeting?</li>
</ul>
```

Our recursive template returns these results:

```
This is a test ^& I hope it works ^| fails gracefully
Some techniques are simpler ^& easier than recursion
Will I enjoy next Tuesday''s meeting?
```

This style of programming takes some getting used to, but whatever you want to do can usually be done. Our example here is a good illustration of the techniques we've discussed in this chapter, including branching statements, variables, invoking templates by name, and passing parameters.

[2.0] Using the XPath 2.0 replace( ) Function to Avoid Recursion

Because string manipulation is a common task in transforming documents, XPath 2.0 provides the very useful `replace( )` function. This lets us provide the original string, the string we want to replace, and the string we want substituted in its place. To review our earlier example, we want to make three replacements in our text:

- Any ampersand (`&`) should have a caret added in front of it (`^&`).
- Any vertical bar (`|`) should have a caret added in front of it (`^|`).
- Any single quote (`'`) should be replaced with two single quotes (`''`).

Our stylesheet to perform these tasks looks like this:

```
<?xml version="1.0"?>
<!-- string_replace-2_0.xsl -->
<xsl:stylesheet version="2.0"
```

```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="text"/>

    <xsl:template match="/">
      <xsl:apply-templates select="ul/li"/>
    </xsl:template>

    <xsl:template match="li">
      <xsl:variable name="sub1" select="replace(., '&amp;', '^&amp;')"/>
      <xsl:variable name="sub2" select="replace($sub1, '\|', '^|')"/>
      <xsl:value-of select='replace($sub2, "&apos;", "&apos;&apos;")'/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:template>

</xsl:stylesheet>
```

We get the same results we got from our much longer XSLT 1.0 stylesheet:

```
This is a test ^& I hope it works ^| fails gracefully
Some techniques are simpler ^& easier than recursion
Will I enjoy next Tuesday''s meeting?
```

This example is far simpler than the recursive technique we used in the XSLT 1.0 stylesheet. Both of them generate the same results, but the code in the XSLT 2.0 stylesheet is much easier to understand.

A quick note about the syntax we used to escape all the single quotes in the XSLT 2.0 stylesheet: our technique was to use the single quote (apostrophe) entity within double quotes. In XPath 2.0, single quotes can be escaped by doubling them in an XPath expression, so we also could have written our call to the `replace( )` function like this:

```
<xsl:value-of select="replace($sub2, '''', '''''')"/>
```

This syntax is really confusing, but it works.

A Stylesheet That Emulates a for Loop

We stressed earlier that the `xsl:for-each` element is not a `for` loop; it's merely an iterator across a group of nodes. However, if you simply must implement a `for` loop, there's a way to do it. (Get ready to use recursion, though.)

Template Design

Our design here is to create a named template that will take some arguments, and then act as a `for` loop processor. If you think about a traditional `for` loop, it has several properties:

One or more initialization statements
: These statements are processed before the `for` loop begins. Typically the initialization statements refer to an *index variable* that is used to determine whether the loop should continue.

An increment statement
: This statement specifies how the index variable should be updated after each pass through the loop.

A boolean expression
: If the expression is `true`, the loop continues; if it is ever `false`, the loop exits.

Let's take a sample from the world of Java and C++:

```
for (int i=0; i<length; i++)
```

In this scintillating example, the initialization statement is `i=0`, the index variable (the variable whose value determines whether we're done or not) is `i`, the boolean expression we use to test whether the loop should continue is `i<length`, and the increment statement is `i++`.

For our purposes here, we're going to make several simplifying assumptions. (Feel free, dear reader, to make the example as complicated as you wish.) Here are the shortcuts we'll take:

- Rather than use an initialization statement, we'll require the caller to set the value of the local variable `i` when it invokes our `for` loop processor. This value is passed as a parameter, so it can be calculated by an XPath expression.
- Rather than specify an increment statement such as `i++`, we'll require the caller to set the value of the local variable `increment`. The default value for this variable is `1`; it can be any negative or positive integer, however. The value of this variable will be added to the current value of `i` after each iteration through our loop.
- Rather than allow any conceivable boolean expression, we'll require the caller to pass in two parameters; `operator` and `testValue`. The allowable values for the `operator` variable are `=`, `<` (coded as `<`), `>` (coded as `>`), `!=`, `<=` (coded as `<=`), and `>=` (coded as `>=`). We're doing things this way because there isn't a way to ask the XSLT processor to evaluate a literal (such as `i<length`) as if it were part of the stylesheet.

Implementation

We'll define four global parameters for our stylesheet:

```
<xsl:param name="i"         select="1"/>
<xsl:param name="increment" select="1"/>
<xsl:param name="operator"  select="'<='"/>
<xsl:param name="testValue" select="10"/>
```

The default values defined here correspond to the C++ or Java statement `for (i = 1; i <= 10; i++)`. We also have a `match="/"` template that invokes our `for` loop processor:

```
<xsl:template match="/">
  <xsl:call-template name="for-loop">
    <xsl:with-param name="i"         select="$i"/>
    <xsl:with-param name="increment" select="$increment"/>
    <xsl:with-param name="operator"  select="$operator"/>
    <xsl:with-param name="testValue" select="$testValue"/>
  </xsl:call-template>
</xsl:template>
```

In the `for-loop` template, our first task is to determine whether the condition is true. We do this by calculating a boolean value with several `<xsl:when>` elements, each of which looks like the one below:

```
<xsl:variable name="testPassed">
  <xsl:choose>
    <xsl:when test="$operator = '!='">
      <xsl:if test="$i != $testValue">
        <xsl:text>true</xsl:text>
      </xsl:if>
    </xsl:when>
    ...
</xsl:variable>
```

If the variable `$testPassed` is `true`, the `for-loop` template calls itself again. Before the `<xsl:call-template>` instruction, we can put whatever logic we want. For our sample, we simply write the current value of `$i` to the output.

The Complete Example

Here's the complete stylesheet:

```
<?xml version="1.0"?>
<!-- for-loop.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:param name="i" select="1"/>
  <xsl:param name="increment" select="1"/>
  <xsl:param name="operator" select="'&lt;='"/>
  <xsl:param name="testValue" select="10"/>

  <xsl:template match="/">
    <xsl:call-template name="for-loop">
      <xsl:with-param name="i"         select="$i"/>
      <xsl:with-param name="increment" select="$increment"/>
      <xsl:with-param name="operator"  select="$operator"/>
      <xsl:with-param name="testValue" select="$testValue"/>
    </xsl:call-template>
  </xsl:template>
```

```
<xsl:template name="for-loop">
  <xsl:param name="i"/>
  <xsl:param name="increment"/>
  <xsl:param name="operator"/>
  <xsl:param name="testValue"/>

  <xsl:variable name="testPassed">
    <xsl:choose>
      <xsl:when test="$operator = '!='">
        <xsl:if test="$i != $testValue">
          <xsl:text>true</xsl:text>
        </xsl:if>
      </xsl:when>
      <xsl:when test="$operator = '&lt;='">
        <xsl:if test="$i &lt;= $testValue">
          <xsl:text>true</xsl:text>
        </xsl:if>
      </xsl:when>
      <xsl:when test="$operator = '&gt;='">
        <xsl:if test="$i &gt;= $testValue">
          <xsl:text>true</xsl:text>
        </xsl:if>
      </xsl:when>
      <xsl:when test="$operator = '='">
        <xsl:if test="$i = $testValue">
          <xsl:text>true</xsl:text>
        </xsl:if>
      </xsl:when>
      <xsl:when test="$operator = '&lt;'">
        <xsl:if test="$i &lt; $testValue">
          <xsl:text>true</xsl:text>
        </xsl:if>
      </xsl:when>
      <xsl:when test="$operator = '&gt;'">
        <xsl:if test="$i &gt; $testValue">
          <xsl:text>true</xsl:text>
        </xsl:if>
      </xsl:when>
      <xsl:otherwise>
        <xsl:message terminate="yes">
          <xsl:text>Sorry, the for-loop emulator only </xsl:text>
          <xsl:text>handles six operators &#xA;</xsl:text>
          <xsl:text>(&lt; | &gt; | = | &lt;= | &gt;= | !=).  </xsl:text>
          <xsl:text>The value </xsl:text>
          <xsl:value-of select="$operator"/>
          <xsl:text> is not allowed.&#xA;</xsl:text>
        </xsl:message>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:variable>

  <xsl:if test="$testPassed='true'">
    <!-- Put your logic here, whatever it might be.  For the purpose      -->
    <!-- of our example, we'll just write some text to the output stream. -->
```

```
      <xsl:text>Value of i=</xsl:text>
      <xsl:value-of select="$i"/>
      <xsl:text>&#xA;</xsl:text>

      <!-- Your logic should end here; don't change the rest of this       -->
      <!-- template!                                                       -->

      <!-- Now for the important part: we increment the index variable and  -->
      <!-- loop.  Notice that we're passing the incremented value, not      -->
      <!-- changing the variable itself.                                    -->

      <xsl:call-template name="for-loop">
        <xsl:with-param name="i"         select="$i + $increment"/>
        <xsl:with-param name="increment" select="$increment"/>
        <xsl:with-param name="operator"  select="$operator"/>
        <xsl:with-param name="testValue" select="$testValue"/>
      </xsl:call-template>
    </xsl:if>
  </xsl:template>

</xsl:stylesheet>
```

Running the stylesheet with the default parameter values creates these exciting results:

```
Value of i=1
Value of i=2
Value of i=3
Value of i=4
Value of i=5
Value of i=6
Value of i=7
Value of i=8
Value of i=9
Value of i=10
```

Using the parameters `i=10`, `increment="-1"`, `operator=">="` and `testValue=0`, we get these results:

```
Value of i=10
Value of i=9
Value of i=8
Value of i=7
Value of i=6
Value of i=5
Value of i=4
Value of i=3
Value of i=2
Value of i=1
Value of i=0
```

The quotes around the values of `increment` and `operator` are necessary when passing those values from the command line to the XSLT processor. As a final test, here are the results for `i=10`, `increment="-2"`, `operator=">"` and `testValue=0`:

```
Value of i=10
Value of i=8
Value of i=6
Value of i=4
Value of i=2
```

If you want to modify the `for` loop to do something useful, put your code between these comments:

```
<!-- Put your logic here, whatever it might be. For the purpose      -->
<!-- of our example, we'll just write some text to the output stream. -->


<!-- Your logic should end here; don't change the rest of this       -->
<!-- template!                                                       -->
```

Summary

We've covered a lot of ground in this chapter, haven't we? We've gone over all of the basic elements you need to add logic and branching to your stylesheets. We discussed some of the similarities between XSLT and other programming languages you might know; more importantly, we discussed how XSLT is different from most of the code you've probably written. In particular, the use of recursion and the principles of variables that don't change takes some getting used to. Despite the learning curve, most of the common tasks you'll need to do will be similar to the exercises we've gone through in this chapter. Now that we've covered these basic elements, we'll talk about links and references, discovering ways to build links between different parts of an XML document.

CHAPTER 6

Creating Links and Cross-References

If you're creating a web site, publishing a book, or processing an XML-based purchase order, chances are many pieces of information will refer to other things. This chapter discusses several ways to link XML elements. It reviews three techniques:

- Using the XML `ID`, `IDREF`, and `IDREFS` datatypes
- Doing more advanced linking with the `key( )` function
- Generating links in unstructured documents

Using the XML ID, IDREF, and IDREFS Datatypes

Our first attempt at linking will be with the XPath `id( )` function. This useful function helps us find the element that has an `ID` attribute with a particular value.

The Datatypes and How They Work

Three of the basic datatypes that are supported by XML Document Type Definitions (DTDs) and XML Schemas are `ID`, `IDREF`, and `IDREFS`. The `ID` and `IDREF` datatypes work according to two rules:

- Every attribute of datatype `ID` must be unique.
- Every value of datatype `IDREF` must match a value of an attribute of datatype `ID` somewhere in the document.

An attribute with a datatype of `IDREFS` contains one or more space-separated values, each of which must match a value of an `ID` elsewhere in the document. The `IDREFS` datatype is a list of `IDREF` values, just as its name implies.

Here is a simple DTD fragment that uses the `ID` and `IDREF` datatypes:

```
<?xml version="1.0"?>
<!-- parts-list1.xml -->
<!DOCTYPE parts-list [
<!ELEMENT parts-list       (component+, part+)>
```

```
<!ELEMENT component        (name, partref+)>
<!ATTLIST component        component-id ID #REQUIRED>

<!ELEMENT name             (#PCDATA)>

<!ELEMENT partref          EMPTY>
<!ATTLIST partref          refid IDREF #REQUIRED>

<!ELEMENT part             (name)>
<!ATTLIST part             part-id ID #REQUIRED>
]>

<parts-list>
...
</parts-list>
```

Here is the XML Schema definition of the same document type:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- parts-list.xsd -->
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="parts-list">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="component" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element ref="part" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="component">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="partref" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="component-id" type="xs:ID" use="required"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="part">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="part-id" type="xs:ID" use="required"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="name" type="xs:string"/>

  <xs:element name="partref">
    <xs:complexType>
      <xs:attribute name="refid" type="xs:IDREF" use="required"/>
```

```
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The DTD and the XML Schema here are semantically identical, so we won't worry about which of the two we use to validate our XML files. (There are many things you can express in XML Schema that aren't possible in DTDs, but in this case the two documents mean the exact same thing.) Here we're using the `ID` and `IDREF` datatypes. We'll take a quick look at the lesser-used `IDREFS` datatype; for our purposes, we process `IDREFS` the same way.

We'll discuss this in more detail later in this chapter, but be aware that you have to validate the XML document for any attributes to be assigned the `ID`, `IDREF`, and `IDREFS` datatypes. If you can't wait to read the details, you can skip ahead to the section "[2.0] The idref() Function" later in this chapter.

To sum up how our parts list document works, a valid `<parts-list>` has one or more `<component>` elements, followed by one or more `<part>` elements. The `<component>` and `<part>` elements are required to have an `ID` attribute (`component-id` or `part-id`, respectively). There's also a `<partref>` element with a required attribute of type `IDREF`; the value of that attribute, named `refid`, must match the value of an `ID` element somewhere in the parts list.

Our first look at linking parts of a document together will use these nicely structured documents.

Linking Parts of an XML Document

To illustrate the value of linking, we'll use a document that defines several `<component>` and `<part>` elements. Each `<component>` uses some number of `<part>`s. Because our XML document contains `ID` and `IDREF` values, we can link different parts of the document together.

Here's how our document looks:

```
<?xml version="1.0"?>
<!-- parts-list1.xml -->
<!DOCTYPE parts-list [
...
]>

<parts-list>
  <component component-id="C28392-33-TT">
    <name>Turnip Twaddler</name>
    <partref refid="P81952-26-PK"/>
    <partref refid="P86679-52-SP"/>
    <partref refid="P81472-68-FD"/>
    <partref refid="P88107-39-GT"/>
```

```
  </component>
...
  <component component-id="C28772-63-OB">
    <name>Olive Bruiser</name>
    <partref refid="P80228-21-PT"/>
    <partref refid="P82387-85-PA"/>
  </component>

  <part part-id="P80228-21-PT">
    <name>Pitter</name>
  </part>
...
  <part part-id="P86994-25-RC">
    <name>Ribbon Curler</name>
  </part>
</parts-list>
```

Our first task will be to look at each `<component>` and list the names of the `<part>`s that it uses. Each of the `refid` attributes of each of the `<partref>` elements refers to the `id` attribute of a `<part>` element.

A Stylesheet That Uses the id( ) Function

Let's look at our desired output. What we want is a simple text document that lists all of the part names for each component, which should look like this:

```
Here is a test of the id() function:

  Turnip Twaddler (component #C28392-33-TT) uses these parts:
    Spanner
    Feather Duster
    Grommet
    Paring Knife

  Prawn Goader (component #C28813-70-PG) uses these parts:
    Paring Knife
    Mucilage
    Ribbon Curler

...

  Olive Bruiser (component #C28772-63-OB) uses these parts:
    Pitter
    Patter
```

The stylesheet to generate these results is pretty straightforward:

```
<?xml version="1.0"?>
<!-- id1.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
```

```
    <xsl:text>&#xA;Here is a test of the id() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:for-each select="/parts-list/component">
      <xsl:text>&#xA;  </xsl:text>
      <xsl:value-of select="name"/>
      <xsl:text> (component #</xsl:text>
      <xsl:value-of select="@component-id"/>
      <xsl:text>) uses these parts:&#xA;    </xsl:text>
      <xsl:for-each select="id(partref/@refid)">
        <xsl:value-of select="name"/>
        <xsl:text>&#xA;    </xsl:text>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

We call the `id( )` function, which returns a node-set/sequence of all the nodes that match all of the `refid` attributes of the `<partref>` elements in each component. Each item in the node-set is the actual `<part>` element, so the XPath expression `select="name"` returns the name of the part (the text of the `<name>` child element).

The `id( )` and `idref( )` functions both have a two-argument version. The second argument for both functions is a node. That parameter tells the XSLT processor to look in the document that contains the node instead of the document that contains the context item. You probably won't need this option, but it's there.

Before we move on to more complicated examples, we'll look at a slightly different XML document, which uses attributes of type `IDREFS` instead of `IDREF`. Here's what it looks like:

```
<?xml version="1.0"?>
<!-- parts-list2.xml -->
<!DOCTYPE parts-list [
<!ELEMENT parts-list       (component+, part+)>

<!ELEMENT component        (name, partref)>
<!ATTLIST component        component-id ID #REQUIRED>

<!ELEMENT name             (#PCDATA)>

<!ELEMENT partref          EMPTY>
<!ATTLIST partref          refid IDREFS #REQUIRED>

<!ELEMENT part             (name) >
<!ATTLIST part             part-id ID #REQUIRED>
]>

<parts-list>
  <component component-id="C28392-33-TT">
    <name>Turnip Twaddler</name>
```

```
    <partref
      refid="P81952-26-PK P86679-52-SP P81472-68-FD P88107-39-GT"/>
  </component>
  <component component-id="C28813-70-PG">
    <name>Prawn Goader</name>
    <partref refid="P81952-26-PK P80499-43-MC P86994-25-RC"/>
  </component>
...
</parts-list>
```

Even though the structure of this XML document is different, we can use the same stylesheet against it. *We get the same results, even though the datatype of the attribute has changed.* When we call the `id( )` function with an argument such as `P81952-PK P86679-52-SP ...`, the `id( )` function treats each space-separated string as a separate ID. The value of any `ID` attribute must be a valid XML name, which means it can't contain spaces. That's why this works. (We'll talk more about valid XML names in a little while.)

We've written a stylesheet that goes from an `IDREF` to the element that has that particular `ID`. Next we'll write a stylesheet that goes from an `ID` to all of the references to it. We'll list each `<part>` in our document, and then list all of the `<component>`s that use it. The challenge is in the XPath expression; given the `ID` of a `<part>`, how do we find all of the `<component>`s that have a `<partref>` element with a `refid` attribute? Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- id2.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Here is a test of the id() </xsl:text>
    <xsl:text>function in reverse:&#xA;</xsl:text>

    <xsl:for-each select="/parts-list/part">
      <xsl:text>&#xA;  </xsl:text>
      <xsl:value-of select="name"/>
      <xsl:text> (part #</xsl:text>
      <xsl:value-of select="@part-id"/>
      <xsl:text>) is used in these products:&#xA;</xsl:text>
      <xsl:for-each
        select="/parts-list/component
                [partref/@refid=current()/@part-id]">
        <xsl:value-of select="name"/>
        <xsl:if test="position() != last()">
          <xsl:text>&#xA;    </xsl:text>
        </xsl:if>
      </xsl:for-each>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
```

```
  </xsl:template>
</xsl:stylesheet>
```

The XPath expression in the <xsl:for-each> is more complicated here; we'll take a closer look at it.

```
/parts-list/component[partref/@refid=current()/@part-id]
```

The expression returns all of the <component> elements for which the predicate expression is true. The predicate expression specifies that the component has at least one <partref> child element whose refid attribute matches the part-id attribute of the current node.

The predicate is comparing two values, even though it looks like a location path with three parts. One of the values is partref/@refid, the set of refid attributes from all the <partref> elements in a given <component>. The other value is current( )/@part-id, the value of the part-id attribute of the current node. Every value inside the predicate refers to a <component> element; the current( ) function refers to the current <part> element we're processing. That <part> element is selected by the first <xsl:for-each> element in the template.

Here are the results of the stylesheet:

```
Here is a test of the id() function in reverse:

  Pitter (part #P80228-21-PT) is used in these products:
    Olive Bruiser

  Patter (part #P82387-85-PA) is used in these products:
    Olive Bruiser

  Spanner (part #P86679-52-SP) is used in these products:
    Turnip Twaddler
    Clam Teaser
    Lemon Snubber

  Feather Duster (part #P81472-68-FD) is used in these products:
    Turnip Twaddler
    Clam Teaser
    Cucumber Decorating Kit
...
```

Inside the inner <xsl:for-each> element, the XPath expression name returns the name of the current <component>. We could have written the inner <xsl:for-each> element like this:

```
<xsl:for-each
  select="/parts-list/component/partref
          [@refid=current()/@part-id]">
  <xsl:value-of select="../name"/>
```

This generates the same results, but the expression to select the name of the component is slightly more complicated. The select attribute of the <xsl:for-each> element returns a node-set of <partref> elements, so we have to use ../name to get the name of

the component. If you find yourself writing lots of complicated expressions inside a <xsl:for-each> element, you should see whether you can rewrite the <for-each> element's XPath expression to simplify your stylesheet.

[2.0] The idref( ) Function

In the previous stylesheet, it was tedious to use the id( ) function in reverse, going from something with a given ID to the elements that reference it. Because this is a fairly common task, XSLT 2.0 adds the idref( ) function. Given an ID, idref( ) returns all of the elements that reference it. Here's a simple stylesheet that works with our parts list:

```
<?xml version="1.0"?>
<!-- idref.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Here is a test of the idref() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:for-each select="/parts-list/part">
      <xsl:text>&#xA;  </xsl:text>
      <xsl:value-of select="name"/>
      <xsl:text> (part #</xsl:text>
      <xsl:value-of select="@part-id"/>
      <xsl:text>) is used in these products:&#xA;</xsl:text>
      <xsl:value-of select="idref(@part-id)/../../name"
        separator="&#xA;"/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

In this stylesheet, we can do everything with a single <xsl:value-of> element. Notice that the idref( ) function returns the matching *attributes*; that's why we use the XPath expression idref(@part-id)/../../name to get the name of the component. The parent of the attribute is a <partref> element and its parent is a <component>. The component's <name> child is what we want here.

Finally, the new idref( ) function works with the IDREFS datatype, just like id( ).

Generating HTML Documents with Links

Before we leave the topic of IDs, we'll look at a more complicated stylesheet—one that generates HTML. We want to list all the components and parts in our document, and we want to create hyperlinks between them. So, we'll list the components and the parts they use; each part name will be a link to a description of that part. We'll do the same

thing when we list each part and the components that use it. Our source document has changed slightly for the purposes of our next few examples:

```
<?xml version="1.0"?>
<!-- parts-list3.xml -->
<!DOCTYPE parts-list [
<!ELEMENT parts-list       (component+, part+, supplier+)>

<!ELEMENT component        (name, partref+, description)>
<!ATTLIST component        component-id ID #REQUIRED>

<!ELEMENT name             (#PCDATA)>

<!ELEMENT partref          EMPTY>
<!ATTLIST partref          refid IDREF #REQUIRED>

<!ELEMENT part             (name, description)>
<!ATTLIST part             part-id ID #REQUIRED
                           supplier CDATA #REQUIRED>

<!ELEMENT description      (#PCDATA|partref)*>

<!ELEMENT supplier         (name)>
<!ATTLIST supplier         country CDATA #REQUIRED
                           vendor-id CDATA #REQUIRED>
]>

<parts-list>
  <component component-id="C28392-33-TT">
    <name>Turnip Twaddler</name>
    <partref refid="P81952-26-PK"/>
    <partref refid="P86679-52-SP"/>
    <partref refid="P81472-68-FD"/>
    <partref refid="P88107-39-GT"/>
    <description>
      If you've got turnips to twaddle, this is the tool for you!
      Comes with a <partref refid="P81472-68-FD"/>.
    </description>
  </component>
  <component component-id="C28100-38-CT">
    <name>Clam Teaser</name>
    <partref refid="P81472-68-FD"/>
    <partref refid="P86994-25-RC"/>
    <partref refid="P86679-52-SP"/>
    <description>
      Everyone knows they're proverbially happy, but what to
      do with a shy clam?  Bring recalcitrant mollusks out of
      their shells with this entertaining gadget.  Includes a
      festive <partref refid="P86994-25-RC"/>.
    </description>
  </component>

...

  <part part-id="P80228-21-PT" supplier="4839">
```

```
    <name>Pitter</name>
    <description>
      Removes pits from olives and cherries in no time at all.
    </description>
  </part>
  <part part-id="P82387-85-PA" supplier="2983">
    <name>Patter</name>
    <description>
      We're not sure what these things do, but people seem
      to like 'em.
    </description>
  </part>

...

  <supplier country="Great Britain" vendor-id="4839">
    <name>Acme Products, Inc.</name>
  </supplier>
  <supplier country="Germany" vendor-id="2983">
    <name>Deutschland Excelsior Gmbh</name>
  </supplier>
  <supplier country="Great Britain" vendor-id="5910">
    <name>Unlimited Spanners Ltd.</name>
  </supplier>
</parts-list>
```

There are a couple of differences here. First of all, we've added a `<description>` element to every `<component>` and `<part>`. To complicate things, some of the descriptions contain a `<partref>` element. Notice that the `<partref>` element doesn't have any text. That means the name of a given part is defined in one place only; if we change the name of a part, the part name will automatically be updated every place we use it. Finally, we've added some `<supplier>` elements and put `country` and `vendor-id` attributes on each part. We'll use those when we talk about keys and key functions.

To generate the HTML document we want, we need to create link points. The description of every component and part should have an HTML anchor (`<a name="...">`), so we can link to those descriptions. Because the parts and components have unique IDs already, we'll use those IDs as the names of the link points. That means we know how to create a link point for each component and part, and we know how to link to a given part or component.

Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- id-html.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
```

```
      <title>Our Catalog</title>
    </head>
    <body style="font-family: sans-serif;">
      <h1>Our Catalog</h1>
      <p>Here's a look at everything in our catalog:</p>
      <h2 style="background: #66FF66;">Components</h2>
      <xsl:apply-templates select="/parts-list/component"/>
      <h2 style="background: #6666FF;">Parts</h2>
      <xsl:apply-templates select="/parts-list/part"/>
    </body>
  </html>
</xsl:template>

<xsl:template match="component">
  <a name="{@component-id}"/>
  <h3>
    <xsl:value-of select="name"/>
  </h3>
  <p>
    <xsl:apply-templates select="description"/>
  </p>
  <p>
    <xsl:value-of select="name"/>
    <xsl:text> uses these parts:</xsl:text>
  </p>
  <ul>
    <xsl:for-each select="partref">
      <li>
        <xsl:apply-templates select="."/>
      </li>
    </xsl:for-each>
  </ul>
</xsl:template>

<xsl:template match="description">
  <xsl:apply-templates select="*|text()"/>
</xsl:template>

<xsl:template match="partref">
  <a href="{concat('#', @refid)}">
    <xsl:value-of select="id(@refid)/name"/>
  </a>
</xsl:template>

<xsl:template match="part">
  <a name="{@part-id}"/>
  <h3>
    <xsl:value-of select="name"/>
  </h3>
  <p>
    <xsl:apply-templates select="description"/>
  </p>
  <p>
    <xsl:value-of select="name"/>
    <xsl:text> is used in these components:</xsl:text>
```

```
    </p>
    <ul>
      <xsl:for-each select="/parts-list/component
                            [partref/@refid=current()/@part-id]">
        <li>
          <a href="{concat('#', @component-id)}">
            <xsl:value-of select="name"/>
          </a>
        </li>
      </xsl:for-each>
    </ul>
  </xsl:template>

</xsl:stylesheet>
```

Here's how we create the link points:

```
<a name="{@part-id}"/>
```

This generates the HTML markup `<a name="C28392-33-TT"/>` for the first `<component>` in the document. Whenever we need to create a link *to* that component, the markup is pretty straightforward:

```
<a href="{concat('#', @component-id)}">
  <xsl:value-of select="name"/>
</a>
```

This generates the HTML markup `<a href="#C28392-33-TT">Turnip Twaddler</a>` for any reference to the Turnip Twaddler. If your document uses `ID`, `IDREF`, and `IDREFS` attributes, creating links with this technique is easy.

Figure 6-1 shows how the HTML document looks.

Limitations of IDs

To this point, we've been able to generate cross-references easily. There are some limitations of the `ID` datatype and the `id( )` function, though:

- If you want to use the `ID` datatype, you have to declare the attributes that use that datatype in your DTD or schema. Unfortunately, if your DTD is defined externally to your XML document, the XML parser isn't required to read it. If the DTD isn't read, then the parser has no idea that a given attribute is of type `ID`. Similarly, if you're using a schema, you have to make sure your XSLT processor validates your XML document against the schema to ensure that the `ID`, `IDREF`, and `IDREFS` datatypes are used correctly.
- You must define the `ID`, `IDREF`, and `IDREFS` relationship in the XML document. It would be nice to have the XML document define the data only, with the relationships between parts of the document defined externally (say, in a stylesheet). That way, if you need to define a new relationship between parts of the document, you could do it by creating a new stylesheet, and you wouldn't have to modify your

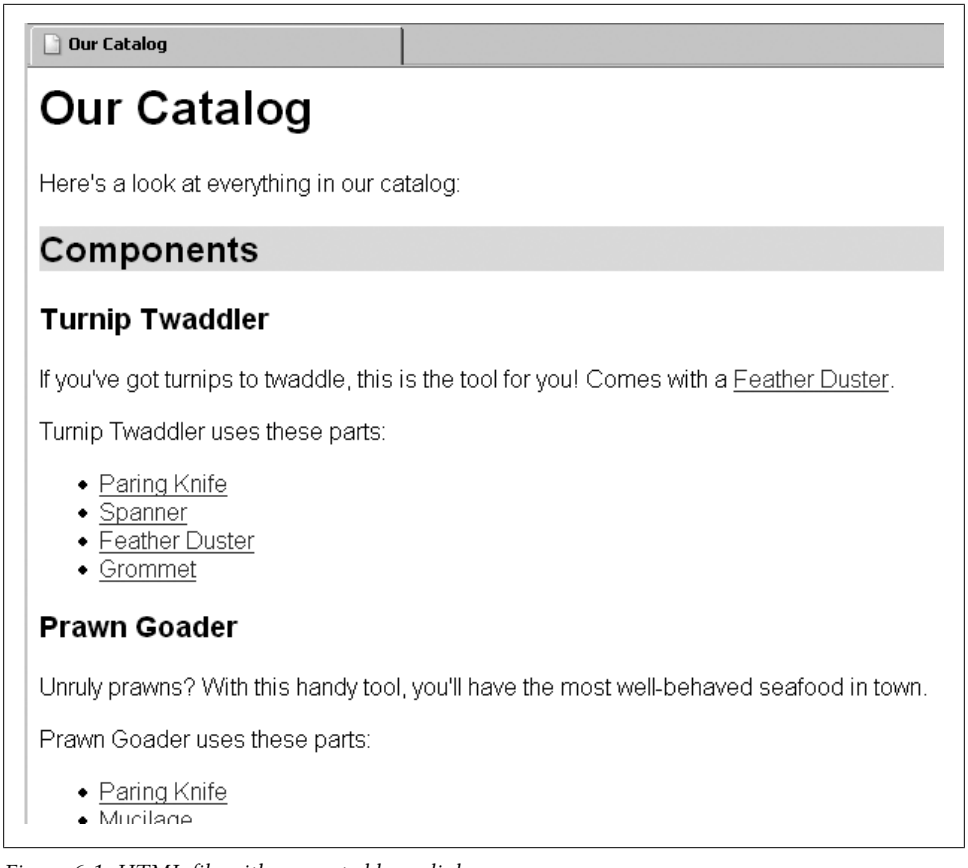

Our Catalog
Our Catalog
Here's a look at everything in our catalog:
Components
Turnip Twaddler
If you've got turnips to twaddle, this is the tool for you! Comes with a Feather Duster.
Turnip Twaddler uses these parts:
Paring Knife
Spanner
Feather Duster
Grommet
Prawn Goader
Unruly prawns? With this handy tool, you'll have the most well-behaved seafood in town.
Prawn Goader uses these parts:
Paring Knife

Figure 6-1. HTML file with generated hyperlinks

XML document. It becomes unwieldy quickly if you have to change the XML document structure every time you need to define a new relationship between parts of the document.

- An element can have at most one attribute of type `ID`. If you'd like to refer to the same element in more than one way, you can't use the `id( )` function.
- Any given `ID` value can be found on one element at most. If you'd like to refer to more than one element with a single value, you can't use the `id( )` function for that, either.
- Only one set of `ID`s exists for the entire document. In other words, if you declare the attributes `component-id` and `part-id` to be of type `ID`, the value of a `component-id` must be unique across all the attributes of type `ID`. It is illegal in this case for a `component-id` to be the same as a `part-id`, even though those attributes might belong to different elements.
- If you're using a DTD, an `ID` can only be an attribute of an XML element. The only way you can use the `id( )` function to refer to another element is through its

attribute of type `ID`. If you want to find another element based on an attribute that isn't an `ID`, or based on the element's content or the element's children, and so on, the `id( )` function is of no use whatsoever.

- If you're using an XML schema, you can define an element with a datatype of `xs:ID`. This means you can use the `id( )` function to find an element. This is an improvement on the situation, but it does require a schema-aware XSLT parser.
- The value of an `ID` must be an XML name. In other words, it can't contain spaces, it can't start with a number, and it's subject to the other restrictions of XML names. (Section 2.3 of the XML Recommendation defines these restrictions; see *http://www.w3.org/TR/REC-xml* if you'd like more information.)

To get around all of these limitations, XSLT defines the `<xsl:key>` element and the `key( )` function. We'll discuss them now.

XSLT's Key Facility

Now that we've covered the `id( )` function in great detail, we'll move on to XSLT's `key( )` function and the `<xsl:key>` element. Each `<xsl:key>` element effectively creates an index of the document. You can then use that index to find all elements that have a particular property. Once the key is created, we can use the `key( )` function to retrieve parts of the document.

For example, if you have a database of (U.S. postal) addresses, you might want to index that database by the people's last names, by the states in which they live, by their zip codes, etc. Each index takes a certain amount of time to build, but it saves processing time later. (Be aware that it can take a significant amount of memory to create a key, particularly for very large documents.) If you want to find all the people who live in the state of Idaho, you can use the index to find all those people directly; you don't have to search the entire database.

We'll discuss the details of how the key facility works, and then we'll compare it to the `id( )` function.

Defining a Key with <xsl:key>

You define a `key( )` function with the `<xsl:key>` element:

```
<xsl:key name="supplier-by-country" match="supplier" use="@country"/>
<xsl:key name="part-by-supplier" match="part" use="@supplier"/>
```

The key has three attributes:

`name`
: This attribute is used to refer to this particular key. When you want to find parts of your XML document, use the `name` to indicate the key you want to use.

`match`
: Containing an XPath expression, this attribute specifies what part of the document you want to index. In our sample here, we've created two keys: one for retrieving `<supplier>`s and one for retrieving `<part>`s.

`use`
: Containing another XPath expression, this attribute is interpreted in the context of the `match` attribute. In other words, the first `<xsl:key>` element here, named `supplier-by-country`, creates an index of all the `<supplier>` elements, and uses the `country` attribute to retrieve them. The second `<xsl:key>` element, named `part-by-supplier`, creates an index of all the `<part>` elements and uses the `supplier` attribute to find them.

[2.0] XSLT 2.0 adds a fourth attribute, `collation`. This allows us to specify a set of rules for how values are compared. To cite a frequent example from the specs, the German word for *street* can be spelled *Strasse* or *Straße*. Using a German collation for the key function causes those to words to be the same, despite the fact that they are clearly different strings.

The XSLT 1.0 specification specifically states that the `match` and `use` attributes can't contain variables.

Generating Links with the key( ) Function

In the modified parts list document we looked at a moment ago, we added a `<supplier>` to each `<part>`. We also added a `country` attribute to the `<supplier>` element. If you look at the document structure as defined in the embedded DTD (or the external schema), you'll notice that the `vendor-id` and `country` attributes of the `<supplier>` element don't have a datatype of `ID`, and that the `supplier` attribute of the `<part>` element is not an `IDREF`.

We want to retrieve all the parts that are provided by a particular country. If we defined the `country` attribute to be of type `ID`, we could only have one supplier from each country. Clearly that's an unacceptable limitation on our document.

Now that we've created a more flexible XML document, we'll use the `key( )` function to process our document. We'll use two keys here. The first retrieves all of the `<supplier>` elements that match a given country name. The second retrieves all of the `<part>` elements whose `supplier` attribute matches a given supplier's ID. Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- key.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
  <xsl:output method="html"/>

  <xsl:param name="country-name"/>

  <xsl:key name="supplier-by-country" match="supplier" use="@country"/>
  <xsl:key name="part-by-supplier" match="part" use="@supplier"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>
          <xsl:text>Parts from </xsl:text>
          <xsl:value-of select="$country-name"/>
        </title>
      </head>
      <body style="font-family: sans-serif;">
        <h1>
          <xsl:text>Parts from </xsl:text>
          <xsl:value-of select="$country-name"/>
        </h1>
        <xsl:choose>
          <xsl:when test="key('supplier-by-country', $country-name)">
            <xsl:apply-templates select="key('supplier-by-country', $country-name)"/>
          </xsl:when>

          <xsl:otherwise>
            <p>Sorry, we don't get any parts from that country!</p>
          </xsl:otherwise>
        </xsl:choose>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="supplier">
    <h2>
      <xsl:value-of select="name"/>
    </h2>
    <p>
      <xsl:value-of select="name"/>
      <xsl:text> supplies these parts:</xsl:text>
    </p>
    <ul>
      <xsl:for-each select="key('part-by-supplier', @vendor-id)">
        <li>
          <b>
            <xsl:value-of select="name"/>
          </b>
          <xsl:text>:  </xsl:text>
          <xsl:apply-templates select="description"/>
        </li>
      </xsl:for-each>
    </ul>
  </xsl:template>

</xsl:stylesheet>
```

Our stylesheet takes an external parameter named `country-name` (defined at the top of the stylesheet) and returns all of the parts supplied by companies based in that country. The first time we use the `key( )` function, we see whether there are any values that match the external parameter:

```
<xsl:when test="key('country-index', $country-name)">
  <xsl:for-each select="key('country-index', $country-name)">
    <xsl:apply-templates select="."/>
  </xsl:for-each>
</xsl:when>
```

The key `country-index` returns all of the `<supplier>` elements that match the given `$country-name`. Assuming there's at least one, we process it (if there's not at least one, the expression evaluates to `false`). Processing the `<supplier>` element uses the other key:

```
<xsl:for-each select="key('part-index', @vendor-id)">
```

This returns all of the `<part>` elements whose `supplier` attribute matches the `vendor-id` attribute of the current `<supplier>`.

Notice that the attribute we're using to retrieve matching nodes can contain spaces. If `country` were of datatype `ID`, it could not have the value `"Great Britain"`. This is one of the advantages of keys. A `country-name` of `"Great Britain"` gives us the HTML document shown in Figure 6-2.

Advantages of the key( ) Function

Now that we've taken the `key( )` function through its paces, you can see that it has several advantages:

- The `key( )` function is defined in a stylesheet. That means I can define any number of relationships between parts of an XML document at any time. If I need to define a new relationship tomorrow, I don't have to change my XML documents.
- Any number of `key( )` functions can be defined for a given element. In our parts list example, we could define `key( )` functions for the values of the `vendor-id`, `part-id`, and `component-id` attributes. We could also create `key( )` functions based on the text of various elements or their children. If we used `ID`s instead of the `key( )` function, we would be limited to a single index based on the value of the single attribute of the `ID` datatype.

  To sum up the advantages for this point, an element can have more than one `key( )` defined against it, and that key doesn't have to be based on an attribute. The key can be based on the element's text, the text of child elements, or other constructs.
- Any number of elements can match a given value. Taking another look at our example, when we use the `key( )` function to find all the parts from a particular country, the `key( )` function returns a node-set that can have any number of nodes.

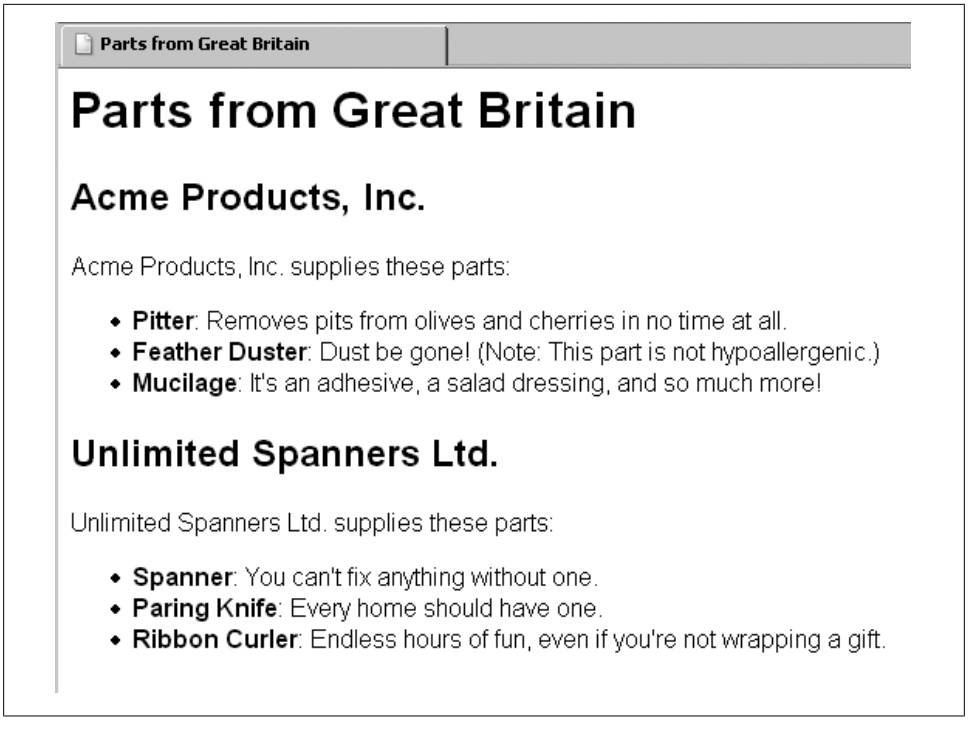

Parts from Great Britain
Parts from Great Britain
Acme Products, Inc.
Acme Products, Inc. supplies these parts:
Pitter: Removes pits from olives and cherries in no time at all.
Feather Duster: Dust be gone! (Note: This part is not hypoallergenic.)
Mucilage: It's an adhesive, a salad dressing, and so much more!
Unlimited Spanners Ltd.
Unlimited Spanners Ltd. supplies these parts:
Spanner: You can't fix anything without one.
Paring Knife: Every home should have one.
Ribbon Curler: Endless hours of fun, even if you're not wrapping a gift.

Figure 6-2. All the parts from our suppliers in Great Britain

If we use an `ID` instead, legally there can be only one element that matches a given country.

- The value we use to look up elements in the key function isn't constrained to be an XML name. If we use the `ID` datatype, its value can't contain spaces, among other constraints.

Normally you'll use the two-argument version of the `key( )` function. We pass the name of the key and the value we're looking for, and the matching items are returned. There's also a three-argument version of the function that lets you limit values returned by the function to a particular set of nodes. See the description of the `key( )` function in Appendix C for more information.

Generating Links in Unstructured Documents

Before we leave the topic of linking, we'll discuss one more useful technique. So far, all of this chapter's examples have been structured nicely. When there was a relationship between two pieces of information, we had an `ID` and `IDREF` pair to match them. What happens if the XML document you're transforming isn't written that way? Fortunately, we can use the `key( )` function and the `generate-id( )` function to create structure where there isn't any.

An Unstructured XML Document in Need of Links

For our example here, we'll take out all of the `id` and `refid` attributes that have served us well so far. This is a contrived example, but it demonstrates how we can use the `key( )` and `generate-id( )` functions to generate links between parts of our document.

In our new sample document, we've stripped out the references that tied things together so neatly before:

```
<?xml version="1.0"?>
<!-- parts-list4.xml -->
<parts-list>
  <component>
    <name>Turnip Twaddler</name>
    <partref>Paring Knife</partref>
    <partref>Spanner</partref>
    <partref>Feather Duster</partref>
    <partref>Grommet</partref>
    <description>
      If you've got turnips to twaddle, this is the tool for you!
      Comes with a <partref>Feather Duster</partref>.
    </description>
  </component>

...

  <part>
    <name>Pitter</name>
    <description>
      Removes pits from olives and cherries in no time at all.
    </description>
  </part>
  <part>
    <name>Patter</name>
    <description>
      We're not sure what these things do, but people seem
      to like 'em.
    </description>
  </part>
...
</parts-list>
```

We've removed all of the `ID`s and `IDREF`s in the document. For elements such as `<partref>` that formerly used attributes to link parts of the document together, we simply use the text of the item we're referring to. To generate the cross-references we created before, we'll need to do three things:

1. Define two keys for all parts and components. One key lets us get the `<part>` that matches a given name, and the other lets us find a `<component>` with a `<partref>` child whose text matches a given name.
2. Generate a new ID for each `<component>` and `<part>` we find.

3. For each <component>, use one key to retrieve the <part> nodes that match a particular name. For each <part>, we use the other key to retrieve the <component> nodes that refer to the current part. We'll use generate-id( ) to create the IDs for us.

We'll go through the relevant parts of the stylesheet. First, we define the two keys we'll use:

```
<xsl:key name="parts-index" match="part/name" use="."/>
<xsl:key name="component-index" match="component" use="partref"/>
```

The first key returns the <name> element that matches a given part name. Notice that the match attribute means we're getting the <name> element; if we want the <part> element itself, we would have to use the parent axis on the node returned by the key( ) function. (We don't need to access the <part> element in our stylesheet; that's why we set up the key this way.)

The second key returns the <component> element that has a <partref> that matches a given part name. What we get from the key( ) function is the <component>, the parent of both the <partref> element that contains the part name we're looking for and the <name> element that we'll want to insert into our HTML document.

The next step is to create an ID for each <component> and <part>:

```
<xsl:template match="component">
  <a name="{generate-id(name)}"/>
...
</xsl:template>
...
<xsl:template match="part">
  <a name="{generate-id(name)}"/>
...
</xsl:template>
```

In both cases, we're generating an ID based on the text of the <name> child of the given element. We're using the names of parts and components throughout our stylesheet, so basing the IDs on the <name> elements makes things simpler.

Now we need to process all of the <partref>s under a given <component>. Here's how that works:

```
<ul>
  <xsl:for-each select="partref">
    <li>
      <a href="{concat('#', generate-id(key('parts-index', .)[1]))}">
        <xsl:value-of select="."/>
      </a>
    </li>
  </xsl:for-each>
</ul>
```

We generate the href attribute of the link by generating an ID of the first match from the key( ) function. The parts-index key returns the <name> element that matches a string; that string in this case is the text value of the current element.

Notice that we used the predicate expression `[1]` to specify the first element from the node-set or sequence. This is good practice for XSLT 1.0, because it makes it clear exactly which node we want. However, this is crucial for XSLT 2.0, because it is a fatal error to pass a sequence with more than one node to the `key( )` function in an XSLT 2.0 stylesheet.

The final task is to create the links from each `<part>` to all of the `<component>`s that use it. Here's how that code looks:

```
<ul>
  <xsl:for-each select="key('component-index', name)">
    <li>
      <a href="{concat('#', generate-id(name))}">
        <xsl:value-of select="name"/>
      </a>
    </li>
  </xsl:for-each>
</ul>
```

Remember, the `component-index` key returns the `<component>` element. The `<component>` is the parent of both the `<partref>` elements and the `<name>` element. The search term we pass to the key is the name of the current part. When we get the `<component>` back from the key, we use its `<name>` child to generate an ID and to write the name of the component.

Here's the complete stylesheet:

```
<?xml version="1.0"?>
<!-- generate-id.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:key name="parts-index" match="part/name" use="."/>
  <xsl:key name="component-index" match="component" use="partref"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Our Catalog</title>
      </head>
      <body style="font-family: sans-serif;">
        <h1>Our Catalog</h1>
        <p>Here's a look at everything in our catalog:</p>
        <h2 style="background: #66FF66;">Components</h2>
        <xsl:apply-templates select="/parts-list/component"/>
        <h2 style="background: #6666FF;">Parts</h2>
        <xsl:apply-templates select="/parts-list/part"/>
      </body>
    </html>
  </xsl:template>
```

```
<xsl:template match="component">
  <a name="{generate-id(name)}"/>
  <h3>
    <xsl:value-of select="name"/>
  </h3>
  <p>
    <xsl:apply-templates select="description"/>
  </p>
  <p>
    <xsl:value-of select="name"/>
    <xsl:text> uses these parts:</xsl:text>
  </p>
  <ul>
    <xsl:for-each select="partref">
      <li>
        <a href="{concat('#', generate-id(key('parts-index', .)[1]))}">
          <xsl:value-of select="."/>
        </a>
      </li>
    </xsl:for-each>
  </ul>
</xsl:template>

<xsl:template match="description">
  <xsl:apply-templates select="*|text()"/>
</xsl:template>

<xsl:template match="component/description/partref">
  <a href="{concat('#', generate-id(key('parts-index', .)[1]))}">
    <xsl:value-of select="."/>
  </a>
</xsl:template>

<xsl:template match="part">
  <a name="{generate-id(name)}"/>
  <h3>
    <xsl:value-of select="name"/>
  </h3>
  <p>
    <xsl:apply-templates select="description"/>
  </p>
  <p>
    <xsl:value-of select="name"/>
    <xsl:text> is used in these components:</xsl:text>
  </p>
  <ul>
    <xsl:for-each select="key('component-index', name)">
      <li>
        <a href="{concat('#', generate-id(name))}">
          <xsl:value-of select="name"/>
        </a>
      </li>
    </xsl:for-each>
  </ul>
```

```
  </xsl:template>

</xsl:stylesheet>
```

Looking at the HTML output in a browser, the document looks exactly the same as our earlier stylesheet. The HTML source code is slightly different, of course:

```
<a name="d1e31"></a>
<h3>Prawn Goader</h3>
<p>
  Unruly prawns?  With this handy tool, you'll have the most
  well-behaved seafood in town.
</p>
<p>Prawn Goader uses these parts:</p>
<ul>
  <li><a href="#d1e181">Paring Knife</a></li>
  <li><a href="#d1e190">Mucilage</a></li>
  <li><a href="#d1e199">Ribbon Curler</a></li>
</ul>
```

All of the names of the anchor points were generated by the XSLT processor. Using a different XSLT processor will probably generate different values for these IDs, but they'll still work. It's even possible that rerunning a document and stylesheet through the same processor will generate different values. (We'll cover all the details of the `generate-id( )` function in the next section.)

Using the `key( )` and `generate-id( )` functions, we've been able to create IDs and references automatically. This approach isn't perfect; we have to make sure the text of the `<partref>` element matches the text of the part's `<name>` exactly. Despite that, `generate-id( )` can help you add some structure to your documents.

The generate-id( ) Function

Before we leave the topic of linking, we'll go over the details of the `generate-id( )` function. This function takes a node-set as its argument, and it works as follows:

- For a given transformation, every time `generate-id( )` is invoked against a given node, it returns the same ID. The ID doesn't change while you're doing a given transformation. If you run the transformation again tomorrow, there's no guarantee that `generate-id( )` will generate the same ID that it generated today. As long as the XSLT processor is running, however, `generate-id( )` returns the same ID for the same node every time.
- If you invoke `generate-id( )` against two different nodes, the two generated IDs will be different.
- *[1.0]* Given a node-set, `generate-id( )` returns an ID for the node in the node-set that occurs first in document order.

  [2.0] It is a fatal error in XSLT 2.0 to pass a sequence of more than one item to `generate-id( )`.

- If the node-set you pass to the function is empty (you invoke `generate-id(fleeber)`, and there are no `<fleeber>` elements in the current context), `generate-id( )` returns an empty string.
- If no node-set is passed in (you invoke `generate-id( )`), the function generates an ID for the context node.

The `generate-id( )` function is not required to check whether an ID it generates duplicates an ID that's already in the document. In other words, if your document has an attribute of type `ID` with a value of `sdk3829a`, there's a possibility that an ID returned by `generate-id( )` will also be `sdk3829a`. It's not likely, but be aware that it could happen.

Summary

In this chapter, we've examined several ways to generate links and cross-references between different parts of a document. If your XML document has a reasonable amount of structure, you can use the `id( )` and `key( )` functions to define many different relationships between the parts of a document. Even if your XML document isn't structured, you may be able to use `key( )` and `generate-id( )` to create simple references. In the next chapter, we'll look at sorting and grouping—two more ways to organize the information in our XML documents.

CHAPTER 7

Sorting and Grouping Elements

By now, I hope you're convinced that you can use XSLT to convert big piles of XML data into other useful things. Our examples to this point have pretty much gone through the XML source in what's referred to as *document order*. We'd like to go through our XML documents in a couple of other common ways, though:

- We could sort some or all of the XML elements, then generate output based on the sorted elements.
- We could group the data, selecting all elements that have some property in common, then sorting the groups of elements.

We'll give several examples of these operations in this chapter.

Sorting Data with <xsl:sort>

The simplest way to rearrange our XML elements is to use the `<xsl:sort>` element. This element temporarily rearranges a collection of elements based on criteria we define in our stylesheet.

Our First Example

For our first example, we'll have a set of U.S. postal addresses that we want to sort. (No chauvinism is intended here; obviously every country has different conventions for mailing addresses. We just needed a short sample document that can be sorted in many useful ways.) Here's our original document:

```
<?xml version="1.0"?>
<!-- names.xml -->
<addressbook>
  <address>
    <name>
      <title>Mr.</title>
      <first-name>Chester Hasbrouck</first-name>
      <last-name>Frisby</last-name>
    </name>
```

```
  <street>1234 Main Street</street>
  <city>Sheboygan</city>
  <state>WI</state>
  <zip>48392</zip>
</address>
<address>
  <name>
    <first-name>Mary</first-name>
    <last-name>Backstayge</last-name>
  </name>
  <street>283 First Avenue</street>
  <city>Skunk Haven</city>
  <state>MA</state>
  <zip>02718</zip>
</address>
<address>
  <name>
    <title>Ms.</title>
    <first-name>Natalie</first-name>
    <last-name>Attired</last-name>
  </name>
  <street>707 Breitling Way</street>
  <city>Winter Harbor</city>
  <state>ME</state>
  <zip>00218</zip>
</address>
<address>
  <name>
    <first-name>Harry</first-name>
    <last-name>Backstayge</last-name>
  </name>
  <street>283 First Avenue</street>
  <city>Skunk Haven</city>
  <state>MA</state>
  <zip>02718</zip>
</address>
<address>
  <name>
    <first-name>Mary</first-name>
    <last-name>McGoon</last-name>
  </name>
  <street>103 Bryant Street</street>
  <city>Boylston</city>
  <state>VA</state>
  <zip>27318</zip>
</address>
<address>
  <name>
    <title>Ms.</title>
    <first-name>Amanda</first-name>
    <last-name>Reckonwith</last-name>
  </name>
  <street>930-A Chestnut Street</street>
  <city>Lynn</city>
  <state>MA</state>
```

```
    <zip>02930</zip>
  </address>
</addressbook>
```

We'd like to generate a list of these addresses, sorted by `<last-name>`. We'll use the magical `<xsl:sort>` element to do the work. Our stylesheet looks like this:

```
<?xml version="1.0"?>
<!-- namesorter1.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:for-each select="addressbook/address">
      <xsl:sort select="name/last-name"/>
      <xsl:if test="name/title">
        <xsl:value-of select="name/title"/>
        <xsl:text> </xsl:text>
      </xsl:if>
      <xsl:value-of select="name/first-name"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="name/last-name"/>
      <xsl:text>&#xA;</xsl:text>
      <xsl:value-of select="street"/>
      <xsl:text>&#xA;</xsl:text>
      <xsl:value-of select="city"/>
      <xsl:text>, </xsl:text>
      <xsl:value-of select="state"/>
      <xsl:text>  </xsl:text>
      <xsl:value-of select="zip"/>
      <xsl:text>&#xA;</xsl:text>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

The heart of our stylesheet is the `<xsl:for-each>` and `<xsl:sort>` elements. The `<xsl:for-each>` element selects the items with which we'll work, and the `<xsl:sort>` element rearranges them before we write them out. (Notice that we use `<xsl:if>` to determine whether a given customer has a courtesy title.)

Notice that we're using `<xsl:output method="text"/>` to generate a text file. (Feel free to generate an HTML file or something more complicated if you want.) Here are the results we get from our first attempt at sorting:

```
Ms. Natalie Attired
707 Breitling Way
Winter Harbor, ME  00218

Mary Backstayge
283 First Avenue
Skunk Haven, MA  02718
```

```
Harry Backstayge
283 First Avenue
Skunk Haven, MA  02718

Mr. Chester Hasbrouck Frisby
1234 Main Street
Sheboygan, WI  48392

Mary McGoon
103 Bryant Street
Boylston, VA  27318

Ms. Amanda Reckonwith
930-A Chestnut Street
Lynn, MA  02930
```

As you can see from the output, the addresses in our original document were sorted by last name. All we had to do was add `<xsl:sort>` to our stylesheet, and all the elements were magically reordered. If you aren't convinced that XSLT can increase your programmer productivity, try writing the Java code and DOM method calls to do the same thing.

We can improve on our stylesheet by sorting addresses by `<first-name>` within `<last-name>`. In our last example, Mary Backstayge should appear after Harry Backstayge. Here's how we can modify our stylesheet to use more than one sort key:

```
<?xml version="1.0"?>
<!-- namesorter2.xsl -->
...
<xsl:template match="/">
  <xsl:for-each select="addressbook/address">
    <xsl:sort select="name/last-name"/>
    <xsl:sort select="name/first-name"/>
    ...
```

We've simply added a second `<xsl:sort>` element to our stylesheet. This element does what we want; it sorts the `<address>` elements by `<first-name>` within `<last-name>`.

Now our output is better:

```
Ms. Natalie Attired
707 Breitling Way
Winter Harbor, ME  00218

Harry Backstayge
283 First Avenue
Skunk Haven, MA  02718

Mary Backstayge
283 First Avenue
Skunk Haven, MA  02718

Mr. Chester Hasbrouck Frisby
1234 Main Street
```

```
Sheboygan, WI  48392

Mary McGoon
103 Bryant Street
Boylston, VA  27318

Ms. Amanda Reckonwith
930-A Chestnut Street
Lynn, MA  02930
```

The Details on the <xsl:sort> Element

Now that we've seen a couple of examples of how `<xsl:sort>` works, we'll go over its syntax, its attributes, and where you can use it.

What's the deal with that syntax?

I'm so glad you asked that question. One thing the XSLT working group could have done is something like this:

```
<xsl:for-each select="addressbook/address" sort-key-1="name/last-name"
  sort-key-2="name/first-name"/>
```

The problem with this approach is that no matter how many `sort-key-x` attributes you define, out of sheer perverseness, someone will cry out that they really need the `sort-key-8293` attribute. To avoid this messy issue, the XSLT designers decided to let you specify the sort keys by using a number of `<xsl:sort>` elements. The first is the primary sort key, the second is the secondary sort key, the 8293rd one is the eight-thousand-two-hundred-and-ninety-third sort key, etc.

Well, that's why the syntax looks the way it does, but how does it actually work? When I first saw this syntax:

```
<xsl:for-each select="addressbook/address">
  <xsl:sort select="name/last-name"/>
  <xsl:sort select="name/first-name"/>
  ...
</xsl:for-each>
```

I thought it meant that all the nodes were sorted during each iteration through the `<xsl:for-each>` element. That seemed incredibly inefficient; if you've sorted all the nodes, why re-sort them each time through the `<xsl:for-each>` element? Actually, the XSLT processor handles all `<xsl:sort>` elements before it does anything, then it processes the `<xsl:for-each>` element as if the `<xsl:sort>` elements weren't there.

It's less efficient, but if it makes you feel better about the syntax, you could write the stylesheet like this:

```
<xsl:template match="/">
  <xsl:for-each select="addressbook/address">
    <xsl:sort select="name/last-name"/>
    <xsl:sort select="name/first-name"/>
```

```
    <xsl:for-each select="."> <!-- This is slower, but it works -->
      <xsl:apply-templates/>
    </xsl:for-each>
  </xsl:for-each>
</xsl:template>
```

(Don't actually do this. I'm only trying to make a point.) This stylesheet generates the same results as our earlier one.

Another approach is to use the `<xsl:sort>` element within `<xsl:apply-templates>`. Here's a stylesheet that does that:

```
<?xml version="1.0"?>
<!-- namesorter3.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:apply-templates select="addressbook/address">
      <xsl:sort select="name/last-name"/>
      <xsl:sort select="name/first-name"/>
    </xsl:apply-templates>
  </xsl:template>

  <xsl:template match="address">
    <xsl:if test="name/title">
      <xsl:value-of select="name/title"/>
      <xsl:text> </xsl:text>
    </xsl:if>
    <xsl:value-of select="name/first-name"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="name/last-name"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="street"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="city"/>
    <xsl:text>, </xsl:text>
    <xsl:value-of select="state"/>
    <xsl:text>  </xsl:text>
    <xsl:value-of select="zip"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

Using `<xsl:sort>` inside `<xsl:apply-templates>` generates the same results as our previous stylesheet.

Attributes

The <xsl:sort> element has several attributes; we'll discuss the most useful ones here. The discussion of the <xsl:sort> element in Appendix A has complete details on all of the attributes.

select
: The select attribute defines the characteristic we'll use for sorting. Its contents is an XPath expression, so you can select elements, text, attributes, comments, ancestors, etc. As always, the XPath expression defined in select is evaluated in terms of the element that contains it. In other words, in this example:

```
<xsl:template match="/">
  <xsl:apply-templates select="addressbook/address">
    <xsl:sort select="name/last-name"/>
    <xsl:sort select="name/first-name"/>
  </xsl:apply-templates>
</xsl:template>
```

: In this example, the select attributes of the <xsl:sort> elements are interpreted from the addressbook/address expression. That means select="name/last-name" refers to a <last-name> element inside a <name> inside an <address> element that's inside an <addressbook> element.

data-type
: The data-type attribute can have three values:

- data-type="text"
- data-type="number"
- A data-type="QName" that identifies a particular datatype. How a given datatype is supported (or if it's supported at all) is implementation-defined.

: The XSLT specification defines the behavior for data-type="text" and data-type="number". Consider this XML document:

```
<?xml version="1.0"?>
<!-- numberlist.xml -->
<numberlist>
  <number>127</number>
  <number>23</number>
  <number>10</number>
</numberlist>
```

: We'll sort these values using the default datatype of text (we could specify data-type="text" to get the same results):

```
<?xml version="1.0"?>
<!-- sort-datatype-text.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>
```

```
  <xsl:template match="/">
    <xsl:for-each select="numberlist/number">
      <xsl:sort select="."/>
      <xsl:value-of select="."/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

When we treat the values of these elements as text, here are the results:

```
10
127
23
```

We get this result because a text-based sort puts anything that starts with a "1" before anything that starts with a "2." If we change the `<xsl:sort>` element to be `<xsl:sort select="." data-type="number"/>`:

```
<?xml version="1.0"?>
<!-- sort-datatype-number1.xsl -->
...
  <xsl:template match="/">
    <xsl:for-each select="numberlist/number">
      <xsl:sort select="." data-type="number"/>
      ...
```

we get these results:

```
10
27
123
```

(See `sort-datatype-number1.xsl` for the complete stylesheet.)

If you use something else here (`data-type="floating-point"`, for example), what the XSLT processor does is anybody's guess. The XSLT specification allows for other values here, but it's up to the XSLT processor to decide how (or if) it wants to process those values. Check your processor's documentation to see whether it does anything relevant or useful for values other than `data-type="text"` or `data-type="number"`.

A final note: if you're using `data-type="number"`, and any of the values aren't numbers, those nonnumeric values will sort before the numeric values. That means that if you're using `order="ascending"`, the nonnumeric values appear first; if you use `order="descending"`, the nonnumeric values appear last.

```
<?xml version="1.0"?>
<!-- badnumberlist.xml -->
<numberlist>
  <number>127</number>
  <number>23</number>
  <number>zzz</number>
  <number>10</number>
```

```
  <number>yyy</number>
</numberlist>
```

Given this less-than-perfect data, here are the correctly sorted results:

```
zzz
yyy
10
23
127
```

Notice that the nonnumeric values were not sorted; they simply appear in the output document in the order in which they were encountered.

[2.0] The `data-type` attribute is deprecated in XSLT 2.0. The preferred way of sorting typed data in XSLT 2.0 is to specify the datatype in the `select` attribute of the `<xsl:sort>` element itself. For example, specifying `<xsl:sort select="xs:integer(.)"/>` forces all of the items that we're sorting to be cast as integers:

```
<?xml version="1.0"?>
<!-- sort-datatype-number2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:for-each select="numberlist/number">
      <xsl:sort select="xs:integer(.)"/>
      <xsl:value-of select="."/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Be aware that using this XSLT 2.0 stylesheet with *badnumberlist.xml* causes a runtime error. Calling `xs:integer('zzz')` doesn't work.

`order`
: You can order the sort as `order="ascending"` or `order="descending"`. The default is `order="ascending"`.

`case-order`
: This attribute can have two values. `case-order="upper-first"` means that uppercase letters sort before lowercase letters, and `case-order="lower-first"` means that lowercase letters sort first. The `case-order` attribute is used only when the `data-type` attribute is `text`. The default value depends on the value of the soon-to-be-discussed `lang` attribute.

`lang`
: This attribute defines the language of the sort keys. The valid values for this attribute are the same as those for the `xml:lang` attribute defined in Section 2.12 of

the XML 1.0 specification. The language codes are those commonly used in Java programming, Unix locales, and other places where ISO language and country namings are defined. For example, `lang="en"` means "English," `lang="en-US"` means "U.S. English," and `lang="en-GB"` means "U.K. English." Without the `lang` attribute (it's rarely used in practice), the XSLT processor determines the default language from the system environment.

Where can you use <xsl:sort>?

The `<xsl:sort>` element can appear inside the `<xsl:apply-templates>` and `<xsl:for-each>` elements.

[2.0] In XSLT 2.0, you can also use `<xsl:sort>` inside the new `<xsl:for-each-group>` and `<xsl:perform-sort>` elements; more on those later in this chapter.

If you use one or more `<xsl:sort>` elements, they must appear first. If you try something like this, you'll get an exception from the XSLT processor:

```
<xsl:for-each select="addressbook/address">
  <xsl:sort select="name/last-name"/>
  <xsl:value-of select="name/title"/>
  <xsl:sort select="name/first-name"/> <!-- NOT LEGAL! -->
  ...
```

Another Example

We've pretty much covered the `<xsl:sort>` element at this point. To add another wrinkle to our example, we'll change the stylesheet so the `xsl:sort` element acts upon a subset of the addresses, and then sorts that subset. We'll sort only the addresses from states that start with the letter `M`. As you'd expect, we'll do this magic with an XPath expression that limits the elements to be sorted:

```
<?xml version="1.0"?>
<!-- namesorter4.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" indent="no"/>

  <xsl:template match="/">
    <xsl:for-each select="addressbook/address[starts-with(state, 'M')]">
      <xsl:sort select="name/last-name"/>
      <xsl:sort select="name/first-name"/>
      <xsl:if test="name/title">
        <xsl:value-of select="name/title"/>
        <xsl:text> </xsl:text>
      </xsl:if>
      <xsl:value-of select="name/first-name"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="name/last-name"/>
      <xsl:text>&#xA;</xsl:text>
      <xsl:value-of select="street"/>
```

```
      <xsl:text>&#xA;</xsl:text>
      <xsl:value-of select="city"/>
      <xsl:text>, </xsl:text>
      <xsl:value-of select="state"/>
      <xsl:text>  </xsl:text>
      <xsl:value-of select="zip"/>
      <xsl:text>&#xA;</xsl:text>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Here are the results—only those addresses from states beginning with the letter `M`, sorted by first name within last name:

```
Ms. Natalie Attired
707 Breitling Way
Winter Harbor, ME  00218

Harry Backstayge
283 First Avenue
Skunk Haven, MA  02718

Mary Backstayge
283 First Avenue
Skunk Haven, MA  02718

Ms. Amanda Reckonwith
930-A Chestnut Street
Lynn, MA  02930
```

Notice that in the `xsl:for-each` element, we used a predicate in our XPath expression so that only addresses containing `<state>` elements whose contents begin with `M` are selected. This example starts us on the path to grouping nodes.

We could do some other things here:

- We could generate output that prints all the unique zip codes, along with the number of addresses that have those zip codes.
- For each unique zip code (or state, or last name, etc.) we could sort on a field and list all addresses with that zip code.

We'll discuss these topics in just a moment. Before we move on to the topic of grouping, we'll go over the new `<xsl:perform-sort>` element.

[2.0] The <xsl:perform-sort> Element

As we discussed in Chapter 3, XSLT 2.0 introduces the concept of a *sequence*, which is a group of nodes or atomic values. That sequence is typically created during stylesheet processing, usually as a variable. You can use the `<xsl:perform-sort>` element to sort a sequence. Everything we've discussed about sorting applies to `<xsl:perform-sort>`; we'll look at some examples here.

There are two ways to use <xsl:perform-sort>: you can give it an existing sequence and use <xsl:perform-sort> to sort that sequence, or you can use <xsl:perform-sort> to both create the sequence and sort it. For our first example, we'll create a sequence of all the <city> elements and use <xsl:perform-sort> to sort it:

```
<?xml version="1.0"?>
<!-- perform-sort1.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="sortedCities" as="xs:string*">
      <xsl:perform-sort select="addressbook/address/city">
        <xsl:sort select="."/>
      </xsl:perform-sort>
    </xsl:variable>
    <xsl:text>Our customers live in these cities:&#xA;&#xA;</xsl:text>
    <xsl:value-of select="$sortedCities" separator="&#xA;"/>
  </xsl:template>

</xsl:stylesheet>
```

The select attribute of <xsl:perform-sort> defines the sequence to be sorted. When we use this stylesheet against our address book, here are the results:

```
Our customers live in these cities:

Boylston
Lynn
Sheboygan
Skunk Haven
Skunk Haven
Winter Harbor
```

We can also use the new <xsl:sequence> element to create the sequence inside <xsl:perform-sort>:

```
<?xml version="1.0"?>
<!-- perform-sort2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="sortedCities" as="xs:string*">
      <xsl:perform-sort>
        <xsl:sort select="."/>
        <xsl:sequence select="addressbook/address/city"/>
      </xsl:perform-sort>
    </xsl:variable>
```

```
    <xsl:text>Our customers live in these cities:&#xA;&#xA;</xsl:text>
    <xsl:value-of select="$sortedCities" separator="&#xA;"/>
  </xsl:template>

</xsl:stylesheet>
```

There's not much point in doing things this way, but this stylesheet produces the same results. Putting the XPath expression on the `<xsl:perform-sort>` is much simpler. The contents of an `<xsl:perform-sort>` element must start with one or more `<xsl:sort>` elements. In our second example, we're not using the `select` attribute, so we use an `<xsl:sequence>` element to select some data to sort. `<xsl:perform-sort>` creates the entire sequence, and then it uses the `<xsl:sort>` elements inside it to sort the sequence.

If we want to make our customer database look more international, we can add more `<xsl:sequence>` elements to select more data:

```
<?xml version="1.0"?>
<!-- perform-sort3.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="sortedCities" as="xs:string*">
      <xsl:perform-sort>
        <xsl:sort select="."/>
        <xsl:apply-templates select="addressbook/address/city"/>
        <xsl:sequence select="('London', 'Adelaide', 'Rome')"/>
        <xsl:sequence select="('Jakarta', 'Sao Paulo', 'Timbuktu')"/>
      </xsl:perform-sort>
    </xsl:variable>
    <xsl:text>Our customers live in these cities:&#xA;&#xA;</xsl:text>
    <xsl:value-of select="$sortedCities" separator="&#xA;"/>
  </xsl:template>

</xsl:stylesheet>
```

Now we have the impressive results we were hoping for:

```
Our customers live in these cities:

Adelaide
Boylston
Jakarta
London
Lynn
Rome
Sao Paulo
Sheboygan
Skunk Haven
Skunk Haven
Timbuktu
Winter Harbor
```

The point of this example is that you can combine multiple sequences of values and have `<xsl:perform-sort>` to sort all of the elements from all of those sequences. In this example, the first sequence is created with `<xsl:apply-templates>`; this uses the built-in stylesheet rules to return the names of the cities. The other two sequences are created with sequences of string values. The combination of `<xsl:perform-sort>` and `<xsl:sort>` produce a single sorted sequence of all the values.

Another important point about `<xsl:perform-sort>`: it always returns a sequence. If the variable `sortedCities` was defined with a datatype of `xs:string` (instead of `xs:string*`), the stylesheet would raise an error if there were no cities, or if there was more than one city.

Finally, if we want to remove the duplicate values from the sequence, we can use the XPath 2.0 function `distinct-values( )`:

```
<?xml version="1.0"?>
<!-- perform-sort4.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="sortedCities" as="xs:string*">
      <xsl:perform-sort>
        <xsl:sort select="."/>
        <xsl:apply-templates select="addressbook/address/city"/>
        <xsl:sequence select="('London', 'Adelaide', 'Rome')"/>
        <xsl:sequence select="('Jakarta', 'Sao Paulo', 'Timbuktu')"/>
      </xsl:perform-sort>
    </xsl:variable>
    <xsl:text>Our customers live in these cities:&#xA;&#xA;</xsl:text>
    <xsl:value-of select="distinct-values($sortedCities)"
      separator="&#xA;"/>
  </xsl:template>

</xsl:stylesheet>
```

Now our results list the lovely town of Skunk Haven once only:

```
Our customers live in these cities:

Adelaide
Boylston
Jakarta
London
Lynn
Rome
Sao Paulo
Sheboygan
Skunk Haven
Timbuktu
Winter Harbor
```

Notice that we call `distinct-values( )` against the entire sequence after it's been generated. If we called `distinct-values( )` on each of the `<xsl:sequence>` elements, that would only eliminate duplicate values in each individual sequence.

Grouping Nodes

When grouping nodes, we sort things to get them into a certain order, and then we group all items that have the same value for the sort key (or keys). We'll use `xsl:sort` for this grouping, and then use variables or functions such as `key( )` or `generate-id( )` to finish the job.

[2.0] XSLT 2.0 has new elements and functions that make grouping much easier. If you're using XSLT 2.0, feel free to skip ahead to the section "[2.0] New Grouping Syntax in XSLT 2.0" later in this chapter.

Our First Attempt

For our first example, we'll take our list of addresses and group them. We'll look for all unique values of the `<zip>` element and list the addresses that match each one. We'll sort the list by zip code, then go through the list. If a given item doesn't match the previous zip code, we'll print out a heading; if it does match, we'll just print out the address. Here's our first attempt:

```
<?xml version="1.0"?>
<!-- namegrouper1.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Addresses grouped by zip code &#xA;</xsl:text>
    <xsl:for-each select="addressbook/address">
      <xsl:sort select="zip"/>
      <xsl:if test="zip!=preceding-sibling::address[1]/zip">
        <xsl:text>&#xA;Zip code </xsl:text>
        <xsl:value-of select="zip"/>
        <xsl:text> (</xsl:text>
        <xsl:value-of select="city"/>
        <xsl:text>, </xsl:text>
        <xsl:value-of select="state"/>
        <xsl:text>): &#xA;</xsl:text>
      </xsl:if>
      <xsl:if test="name/title">
        <xsl:value-of select="name/title"/>
        <xsl:text> </xsl:text>
      </xsl:if>
      <xsl:value-of select="name/first-name"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="name/last-name"/>
```

```
      <xsl:text>&#xA;</xsl:text>
      <xsl:value-of select="street"/>
      <xsl:text>&#xA;&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Our approach in this stylesheet consists of two steps:

1. Sort the addresses by zip code:

   ```
   <xsl:sort select="zip"/>
   ```

2. For each address, if its zip code doesn't match the previous one, print out a heading, and then print out the addresses that match it:

   ```
   <xsl:if test="zip!=preceding-sibling::address[1]/zip">
     <xsl:text>&#xA;Zip code </xsl:text>
   ...
   ```

   (Remember that `preceding-sibling` returns a `NodeSet`/`Sequence`, so `preceding-sibling::address[1]` represents the first preceding sibling.)

That sounds reasonable, doesn't it? Let's take a look at the results:

```
Addresses grouped by zip code

Zip code 00218 (Winter Harbor, ME):
Ms. Natalie Attired
707 Breitling Way


Zip code 02718 (Skunk Haven, MA):
Mary Backstayge
283 First Avenue


Zip code 02718 (Skunk Haven, MA):
Harry Backstayge
283 First Avenue


Zip code 02930 (Lynn, MA):
Ms. Amanda Reckonwith
930-A Chestnut Street


Zip code 27318 (Boylston, VA):
Mary McGoon
103 Bryant Street

Mr. Chester Hasbrouck Frisby
1234 Main Street
```

Yes, that certainly seemed like a good approach, but there's one major problem: *it doesn't work*.

Looking at our results, there are two things wrong: one of the addresses (Mr. Chester Hasbrouck Frisby) is incorrectly grouped under the heading for Boylston, Virginia, and there are two groups for Skunk Haven, Massachusetts. The problem here is that the axes work with the *document order*, not the sorted order we've created inside the `<xsl:for-each>` element.

As straightforward as our logic seemed, we'll have to find another way.

A Brute-Force Approach

One thing we could do is make the transformation in two passes; we could write an intermediate stylesheet to sort the names and generate a new XML document, and then use the stylesheet we've already written, because document order and sorted order will be the same. Here's how that intermediate stylesheet would look:

```
<?xml version="1.0"?>
<!-- namegrouper2a.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="no"/>

  <xsl:strip-space elements="*"/>

  <xsl:template match="/">
    <addressbook>
      <xsl:for-each select="addressbook/address">
        <xsl:sort select="name/zip"/>
        <xsl:copy-of select="."/>
      </xsl:for-each>
    </addressbook>
  </xsl:template>
</xsl:stylesheet>
```

This stylesheet generates a new `<addressbook>` document that has all of the `<address>` elements sorted correctly. We can then run our original stylesheet against the sorted document and get results that are closer to what we want:

```
Addresses grouped by zip code
Ms. Natalie Attired
707 Breitling Way


Zip code 02718 (Skunk Haven, MA):
Harry Backstayge
283 First Avenue

Mary Backstayge
283 First Avenue


Zip code 02930 (Lynn, MA):
Ms. Amanda Reckonwith
```

```
930-A Chestnut Street


Zip code 27318 (Boylston, VA):
Mary McGoon
103 Bryant Street


Zip code 48392 (Sheboygan, WI):
Mr. Chester Hasbrouck Frisby
1234 Main Street
```

There's one more problem here: we don't have a heading for the first group. Natalie Attired lives in Winter Harbor, Maine, but there's no heading for Winter Harbor. The answer is to change our XPath expression slightly to see whether this is the first `<address>` element:

```
<?xml version="1.0"?>
<!-- namegrouper2b.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" indent="no"/>

  <xsl:template match="/">
    <xsl:text>Addresses grouped by zip code &#xA;</xsl:text>
    <xsl:for-each select="addressbook/address">
      <xsl:sort select="zip"/>
      <xsl:if test="position() = 1 or
                    zip!=preceding-sibling::address[1]/zip">
        <xsl:text>&#xA;Zip code </xsl:text>
        <xsl:value-of select="zip"/>
...
```

If this is the first `<address>` element, there is no `preceding-sibling`. Our test condition always returns `false`, and the heading for the first address never prints.

Our pair of stylesheets works, but it's not very elegant. Even worse, it's really slow because we have to stop in the middle and write a file out to disk, then read that data back in. We'll find a way to group elements in a single stylesheet, but we'll have to do it with a different technique.

Grouping with <xsl:variable>

We mentioned earlier that sometimes `<xsl:variable>` is useful for grouping, so let's try that approach. We'll save the value of the `<zip>` element each time through the `<xsl:for-each>` element and use `preceding-sibling` in a slightly different way. Here's how attempt number three looks:

```
<?xml version="1.0"?>
<!-- namegrouper3.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
  <xsl:output method="text" indent="no"/>

  <xsl:template match="/">
    <xsl:text>Addresses sorted by zip code&#xA;</xsl:text>
    <xsl:for-each select="addressbook/address">
      <xsl:sort select="zip"/>
      <xsl:variable name="lastZip" select="zip"/>
      <xsl:if test="not(preceding-sibling::address[zip=$lastZip])">
        <xsl:text>Zip code </xsl:text>
        <xsl:value-of select="zip"/>
        <xsl:text>: &#xA;</xsl:text>
        <xsl:for-each select="/addressbook/address[zip=$lastZip]">
          <xsl:sort select="name/last-name"/>
          <xsl:sort select="name/first-name"/>
          <xsl:if test="name/title">
            <xsl:value-of select="name/title"/>
            <xsl:text> </xsl:text>
          </xsl:if>
          <xsl:value-of select="name/first-name"/>
          <xsl:text> </xsl:text>
          <xsl:value-of select="name/last-name"/>
          <xsl:text>&#xA;</xsl:text>
          <xsl:value-of select="street"/>
          <xsl:text>&#xA;&#xA;</xsl:text>
        </xsl:for-each>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

This stylesheet generates what we want:

```
Addresses sorted by Zip Code
Zip code 00218:
Ms. Natalie Attired
707 Breitling Way

Zip code 02718:
Harry Backstayge
283 First Avenue

Mary Backstayge
283 First Avenue

Zip code 02930:
Ms. Amanda Reckonwith
930-A Chestnut Street

Zip code 27318:
Mary McGoon
103 Bryant Street

Zip code 48392:
Mr. Chester Hasbrouck Frisby
1234 Main Street
```

So why does this approach work when our first attempt didn't? The answer is that we don't count on the sorted order of the elements to generate the output. The downside of this approach is that we go through several steps to get the results we want:

1. We sort all the addresses by zip code:

```
<xsl:sort select="zip"/>
```

2. We store the current `<zip>` element's value in the variable `lastZip`:

```
<xsl:variable name="lastZip" select="zip"/>
```

3. For each `<zip>` element, we look at all of its preceding siblings to see whether this is the first time we've encountered this particular value (stored in `lastZip`). If it is, there won't be any preceding siblings that match.

```
<xsl:if test="not(preceding-sibling::address[zip=$lastZip])">
```

4. If this is the first time we've encountered this value in the `<zip>` element, we go back and reselect all `<address>` elements with `<zip>` children that match this value. Once we have that group, we sort them by first name within last name and print each address.

```
<xsl:for-each select="/addressbook/address[zip=$lastZip]">
  <xsl:sort select="name/last-name"/>
  <xsl:sort select="name/first-name"/>
```

So, we've found a way to get the results we want, but it's really inefficient. We sort the data, then we look at each zip code in sorted order, then see whether we've encountered that value before in document order, and then we reselect all the items that match the current zip code and resort them before we write them out.

This has reasonable performance when we're grouping the six elements in our sample document, but the amount of work the XSLT processor has to do increases exponentially as the number of elements we're grouping increases. There's got to be a better way, right? Actually there is, and we discuss it in the next section. Read on....

The <xsl:key> Approach

In this section, we'll look at using `<xsl:key>` to group items in an XML document. This approach is commonly referred to as the "Muench method," after Oracle XML Evangelist (and O'Reilly author) Steve Muench, who first suggested this technique. The Muench method has three steps:

1. Define a `key` for the property we want to use for grouping.
2. Select all of the nodes we want to group. We'll do some tricks with the `key( )` and `generate-id( )` functions to find the unique grouping values.
3. For each unique grouping value, use the `key( )` function to retrieve all nodes that match it. We can further sort those nodes if we want.

Well, that's how the technique works—let's start building the stylesheet that makes the magic happen. The first step, creating a key function, is easy. Here's how it looks:

```
<xsl:key name="zipcodes" match="address" use="zip"/>
```

This `<xsl:key>` element defines a new index called `zipcodes`. It indexes `<address>` elements based on the value of the `<zip>` element they contain.

Now that we've defined our `key`, we're ready for the complicated part. We use the `key( )` and `generate-id( )` functions together. Here's the syntax, which we'll discuss extensively in a minute:

```
<xsl:for-each select="//address[generate-id(.)=
  generate-id(key('zipcodes', zip)[1])]">
```

OK, let's start digging through this syntax. We're selecting all `<address>` elements in which the automatically generated `id` matches the automatically generated `id` of the first node returned by the `key( )` function when we ask for all `<address>` elements that match the current `<zip>` element.

Well, that's clear as crystal, isn't it? Let me try to explain that again from a slightly different perspective.

For each `<address>`, we use the `key( )` function to retrieve all `<address>`es that have the same `<zip>`. We then take the first node from that node-set. Finally, we use the `generate-id( )` function to generate an `id` for both nodes. If the two generated `id`s are identical, then the two nodes are the same.

Whew. Let me catch my breath.

If this `<address>` matches the first node returned by the `key( )` function, then we know we've found the first `<address>` that matches this grouping value. Selecting all of the first values (remember, our previous predicate ends with `[1]`) gives us a node-set of some number of `<address>` elements, each of which contains one of the unique grouping values we need.

That's how the Muench method works. At this point, we've got a way to generate a node-set that contains all of the unique grouping values; now we need to process those nodes. From this point, we'll do several things, all of which are comparatively simple:

1. Sort all nodes based on the grouping property. In this example, the property is the `<zip>` element. We start by selecting the first occurrence of every unique `<zip>` element in the document, and then we sort those `<zip>` elements. Here's how it looks in the stylesheet:

   ```
   <xsl:for-each
     select="//address[generate-id(.)=generate-id(key('zipcodes', zip)[1])]">
     <xsl:sort select="zip"/>
   ```

2. The outer `<xsl:for-each>` element selects all the unique values of the `<zip>` element. Next, we use the `key( )` function to retrieve all `<address>` elements that match the current `<zip>` element:

```
    <xsl:for-each select="key('zipcodes', zip)">
```

3. The `key( )` function gives us a node-set of all matching `<address>` elements. For each group, we sort the group based on the `<last-name>` and `<first-name>` elements, print the heading, and then print each address.

To improve the looks of our output, our final stylesheet will use the techniques we've been building to create an HTML file. Here's the complete listing:

```
<?xml version="1.0"?>
<!-- namegrouper4.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:key name="zipcodes" match="address" use="zip"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Customers grouped by Zip code</title>
      </head>
      <body style="font-family: sans-serif;">
        <table border="1" cellpadding="5">
          <xsl:for-each select="//address[generate-id(.)=
                                generate-id(key('zipcodes', zip)[1])]">
            <xsl:sort select="zip"/>
            <xsl:for-each select="key('zipcodes', zip)">
              <xsl:sort select="name/last-name"/>
              <xsl:sort select="name/first-name"/>
              <tr>
                <xsl:if test="position() = 1">
                  <td style="background: #66FF66; text-align: center;
                             vertical-align: middle; font-weight: bold;"
                    rowspan="{count(key('zipcodes', zip))}">
                    <xsl:text>Zip code </xsl:text>
                    <br/>
                    <span style="font-size: 150%;">
                      <xsl:value-of select="zip"/>
                    </span>
                    <br/>
                    <xsl:value-of select="city"/>
                    <xsl:text>, </xsl:text>
                    <xsl:value-of select="state"/>
                  </td>
                </xsl:if>
                <td style="text-align: right; vertical-align: middle;">
                  <xsl:value-of select="name/first-name"/>
                  <xsl:text> </xsl:text>
                  <span style="font-weight: bold; font-size: 125%;">
                    <xsl:value-of select="name/last-name"/>
                  </span>
                </td>
                <td>
                  <xsl:value-of select="street"/>
```

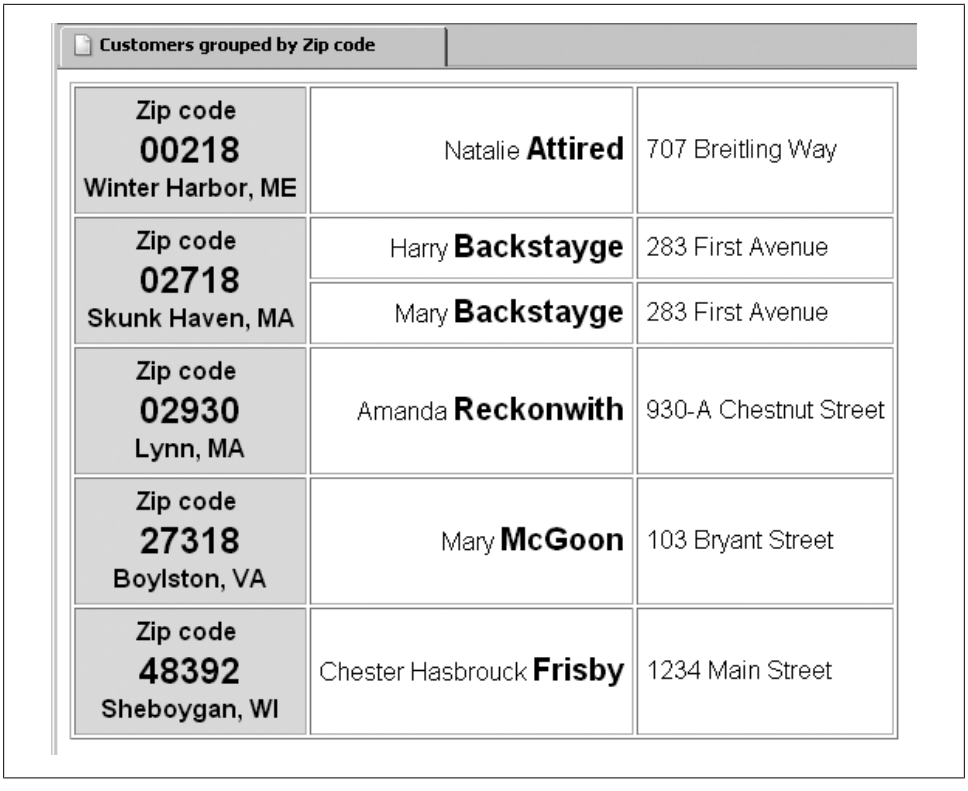


Figure 7-1. HTML document with grouped items

```
                </td>
              </tr>
            </xsl:for-each>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

Notice how the two `<xsl:for-each>` and the various `<xsl:sort>` elements work together. The outer `<xsl:for-each>` element selects the unique values of the `<zip>` element and sorts them; the inner `<xsl:for-each>` element selects all `<address>` elements that match the current `<zip>` element, and then sorts them by `<last-name>` and `<first-name>`.

When we view the generated HTML document in a browser, it looks like Figure 7-1.

The left column contains a table cell for each zip code. That means we need a `rowspan` attribute based on the size of the current group. The logic looks like this:

```
<xsl:if test="position() = 1">
  <td valign="center" bgcolor="#999999"
```

```
        rowspan="{count(key('zipcodes', zip))}">
        <b>
          <xsl:text>Zip code </xsl:text>
          <xsl:value-of select="zip"/>
        </b>
      </td>
    </xsl:if>
```

[2.0] New Grouping Syntax in XSLT 2.0

In 2001, the XSL Working Group released a document entitled "XSLT Requirements Version 2.0." More than half of the document came under the heading *"Must Simplify Grouping."* (I can't imagine how the group would have met a requirement named *"Must Make Grouping More Complicated and Confusing."*) We'll take a look at those changes in this section.

XSLT 2.0's grouping functions are built around the `<xsl:for-each-group>` element. Within this element, we'll use the new XSLT functions `current-group( )` and `current-grouping-key( )` to work with the data we're grouping. There are four mutually exclusive attributes for the `<xsl:for-each-group>` element, each of which performs a different style of grouping:

`group-by`
: This is the most common type of grouping. We use an XPath expression to define what identifies a group (all of the `<address>` elements that have the same `<zip>` code, for example), so we can then iterate through each group.

`group-adjacent`
: This approach is useful when you want to build a group containing all the adjacent nodes that match an XPath expression. As an example, we'll take all the adjacent `<p>` elements in a document, convert each one to an `<li>` element, and put `<ul>` and `</ul>` tags around the entire group.

`group-starting-with`
: The `group-starting-with` attribute defines an XPath expression that identifies the start of a group. Once a group starts, every element is added to the group until the start of another group is found. `group-starting-with` and `group-ending-with` are most often used when adding structure to an HTML document.

`group-ending-with`
: This defines an XPath expression that identifies the end of a group. When using `group-ending-with`, `<xsl:for-each-group>` creates a new group as it begins processing nodes. Whenever the end of a group is found, the XSLT processor closes that group and starts another.

Keep in mind that everything you can do with grouping in XSLT 2.0 is possible in XSLT 1.0—it's just that the markup you'll have to write and maintain in XSLT 1.0 is much more complicated. If you already have a working XSLT 1.0 stylesheet that uses the

Muench method to do grouping, there's no reason to change the stylesheet if you don't want to.

The Most Common Grouping Style: group-by

As we just mentioned, everything we'll do with grouping in XSLT 2.0 revolves around the new `<xsl:for-each-group>` element. This works much like `<xsl:for-each>`. In `<xsl:for-each>`, in each iteration we process an item in a sequence; with `<xsl:for-each-group>`, in each iteration we process a group. Thinking back to our addresses example, each group represents a unique zip code. When using `<xsl:for-each-group>`, the first group would be 00218, the second group would be 02718, and so forth.

To repeat from our discussion of the Muench method, we needed to do three things to group items in XSLT 1.0:

1. Define a `key` for the property we want to use for grouping.
2. Select all of the nodes we want to group. We'll do some tricks with the `key( )` and `generate-id( )` functions to find the unique grouping values.
3. For each unique grouping value, use the `key( )` function to retrieve all nodes that match it. Because the `key( )` function returns a node-set, we can do further sorts on the set of nodes that match any given grouping value.

With XSLT 2.0, we need to do the same basic things, but we don't have to get bogged down with `key( )` and `generate-id( )`. Our tasks are as follows:

1. Define an XPath expression for the property we want to use for grouping. All we have to do is define the XPath expression—we don't need a `key`.
2. Select all of the nodes we want to group. We select all the nodes with an XPath expression. The XSLT processor takes all the items that match this expression and groups them using the grouping property we defined.
3. Instead of dealing with each unique value of the property we're using for grouping, we use `current-group( )` to deal with each group. The `current-group( )` function returns a sequence, so we can sort the members of each group however we like. If we need the value of the grouping key, the `current-grouping-key( )` function does what we want.

Accomplishing these tasks in XSLT 2.0 is pretty straightforward. We do steps one and two with the `<xsl:for-each-group>` element:

```
<xsl:for-each-group select="//address" group-by="zip">
```

The elements we're grouping are all of the `<address>` elements in the document. The property we're using for grouping is the value of the `<zip>` element.

For the third step, we use `current-group( )` to deal with each group in turn. Here's the complete stylesheet:

```
<?xml version="1.0"?>
<!-- for-each-group_group-by.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" include-content-type="no"/>

  <xsl:template match="/">
    <table border="1" cellpadding="5" style="font-family: sans-serif;">
      <xsl:for-each-group select="//address" group-by="zip">
        <xsl:sort select="current-grouping-key()"/>
        <xsl:for-each select="current-group()">
          <xsl:sort select="name/last-name"/>
          <xsl:sort select="name/first-name"/>
          <tr>
            <xsl:if test="position() = 1">
              <td valign="center" bgcolor="#999999"
                rowspan="{count(current-group())}">
                <b>
                  <xsl:text>Zip code </xsl:text>
                  <xsl:value-of select="current-grouping-key()"/>
                </b>
              </td>
            </xsl:if>
            <td align="right">
              <xsl:value-of select="name/first-name"/>
              <xsl:text> </xsl:text>
              <b><xsl:value-of select="name/last-name"/></b>
            </td>
            <td>
              <xsl:value-of select="street"/>
              <xsl:text>, </xsl:text>
              <xsl:value-of select="city"/>
              <xsl:text>, </xsl:text>
              <xsl:value-of select="state"/>
              <xsl:text> </xsl:text>
              <xsl:value-of select="zip"/>
            </td>
          </tr>
        </xsl:for-each>
      </xsl:for-each-group>
    </table>
  </xsl:template>

</xsl:stylesheet>
```

There are several things worth discussing inside the `<xsl:for-each-group>` element:

1. First of all, we use `<xsl:sort select="current-grouping-key( )"/>` to put the groups themselves in order. This `<xsl:sort>` element sorts the *groups* based on their grouping keys. The items inside each group aren't rearranged at all.
2. Next, we use `<xsl:for-each select="current-group">` to iterate through each item in the current group. (Remember, `current-group( )` returns the sequence of nodes that make up the current group.)

3. Within each group, we sort things by `<last-name>` and `<first-name>`.
4. The last significant thing we do is create the first column to display the current grouping key. We get the current value of the grouping key with the `current-grouping-key( )` function. To calculate the `rowspan` attribute, we use an attribute value template with `count(current-group( ))` to get the number of nodes that match the current grouping key.

Comparing this stylesheet with the Muench method version, most of the code is the same; after all, we're building the same HTML document here. However, there are some differences that make the v2.0 stylesheet simpler to create and maintain:

- It's much easier to specify what we're grouping. In the v2.0 stylesheet, we're simply specify the element we're grouping:

  ```
  [2.0] select="//address"

  [1.0] select="//address[generate-id(.)=generate-id(key('zipcodes', zip)[1])]">
  ```

- It's much easier to keep up with the current group. In the v2.0 stylesheet, the `current-group( )` and `current-grouping-key( )` functions let us work with the current group and the value we used to create it. In the v1.0 stylesheet, we have to respecify them each time. Here's how we process all the elements in the current group:

  ```
  [2.0] <xsl:for-each select="current-group()">

  [1.0] <xsl:for-each select="key('zipcodes', zip)">
  ```

  As another example, finding the size of the current group is much simpler. In the v1.0 stylesheet, we have to redefine what the current group is each time we refer to it. When we need the number of items in the current group, we have to do the same thing:

  ```
  [2.0] rowspan="{count(current-group()}"

  [1.0] rowspan="{count(key('zipcodes', zip))}"
  ```

  It's conceptually simpler to get the value of the grouping key in the v2.0 stylesheet, although it's arguable how much of an advantage that actually is in this example. (Comparing the two stylesheets, we typed `<xsl:value-of select="current-grouping-key( )"/>` in v2.0, but only `<xsl:value-of select="zip"/>` in v1.0.) If the key value were much more complicated, the advantage of `current-grouping-key( )` would be greater.

  Also keep in mind that if we change the grouping key in a v1.0 stylesheet, we have to find all the instances of the grouping key and change them. In a v2.0 stylesheet, the grouping key is defined in one place; we still use `current-grouping-key( )` regardless of the changes to the key. This simplifies maintenance.

- We don't have to define an `<xsl:key>` separate from the grouping code. That's a minor point, but it does simplify maintenance.

Another Type of Grouping: group-adjacent

With the `group-adjacent` approach, we'll create a group based on some number of elements that are together in the source document. Our example input document is an HTML document that features groups of paragraphs together:

```
<?xml version="1.0"?>
<!-- group-adjacent_input.html -->
<html>
  <body>
    <h2>Steps for grouping in the Muench method</h2>
    <p>Define a <code>key</code> for the property we want
    to use for grouping.</p>
    <p>Select all of the nodes ...</p>
    <p>For each unique grouping value, ...</p>
    <h2>Steps for grouping in XSLT 2.0</h2>
    <p>Define an XPath expression ...</p>
    <p>Select all of the nodes we want to group ...</p>
    <p>Instead of dealing with each ...</p>
  </body>
</html>
```

This is text from earlier in this chapter, displayed here as HTML. (This book is written entirely in DocBook, an XML vocabulary with a very well-defined structure.) What we want to do is convert any sequence of paragraphs into an unordered list (`<ul>`), with each paragraph converted into a list item (`<li>`). We'll use `group-adjacent` to do that. Our stylesheet looks like this:

```
<?xml version="1.0"?>
<!-- for-each-group_group-adjacent.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" include-content-type="no"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Grouping with group-adjacent</title>
      </head>
      <body style="font-family: sans-serif;">
        <h1>Grouping with group-adjacent</h1>
        <xsl:for-each-group select="html/body/*"
          group-adjacent="boolean(self::p)">
          <xsl:choose>
            <xsl:when test="current-grouping-key()">
              <ul>
                <xsl:for-each select="current-group()">
                  <li>
                    <xsl:copy-of select="@*"/>
                    <xsl:apply-templates select="*|text()"/>
                  </li>
                </xsl:for-each>
              </ul>
```

```
            </xsl:when>
            <xsl:otherwise>
              <xsl:for-each select="current-group()">
                <xsl:apply-templates select="."/>
              </xsl:for-each>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:for-each-group>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="*">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

When we're using `group-adjacent`, the value of the `group-adjacent` attribute generates an *atomic value* for each item in the list. In other words, the grouping key for each item is defined by the `group-adjacent` attribute. In the previous example, any `<p>` element has a grouping key of `true`; everything else has a grouping key of `false`. After we've defined the grouping key for each item, the XSLT 2.0 processor creates the groups. Each group contains the maximum number of adjacent items with a particular grouping key. To see the grouping keys, we can add an `<xsl:message>` to display them. We'll add this message:

```
<xsl:for-each-group select="html/body/*"
  group-adjacent="boolean(self::p)">
  <xsl:message terminate="no">
    <xsl:for-each select="current-group()">
      <xsl:text>current-grouping-key() = </xsl:text>
      <xsl:value-of select="current-grouping-key()"/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:message>
  ...
```

When we run the stylesheet with our sample document, here's what shows up in the console:

```
current-grouping-key() = false

current-grouping-key() = true
current-grouping-key() = true
current-grouping-key() = true

current-grouping-key() = false

current-grouping-key() = true
```

```
current-grouping-key() = true
current-grouping-key() = true
```

For each group, we print out the grouping key for each item in that group. As you'd expect, we have four groups. The first group contains the <h2> element at the start of the document, and the second group contains the three adjacent <p> elements that follow it. The second <h2> is the third group, and the fourth group is the three adjacent <p> elements at the end. When we run the stylesheet, the generated HTML document looks like this:

```
<html>
  <head>
    <title>Grouping with group-adjacent</title>
  </head>
  <body>
    <h1>Grouping with group-adjacent</h1>
    <h2>Steps for grouping in the Muench method</h2>
    <ul>
      <li>Define a <code>key</code> for the property we want
        to use for grouping.</li>
      <li>Select all of the nodes. ...</li>
      <li>For each unique grouping value, ...</li>
    </ul>
    <h2>Steps for grouping in XSLT 2.0</h2>
    <ul>
      <li>Define an XPath expression ...</li>
      <li>Select all of the nodes we want to group ...</li>
      <li>Instead of dealing with each ...</li>
    </ul>
  </body>
</html>
```

The items are now grouped just the way we want. We've converted all of the adjacent <p> elements and replaced them with an unordered list in which each <p> is now a list item. Each group contains the maximum number of adjacent <p> elements, so we can put each group inside a <ul> element.

Before we move on to the final two ways of grouping, we'll look at a more advanced example. We'll change our source HTML document slightly to add some <p> elements that should be processed differently. Here's how our new HTML document looks:

```
<?xml version="1.0"?>
<!-- group-adjacent_input2.html -->
<html>
  <body>
    <!-- Here's some sample text from the chapter "Sorting
         and Grouping". -->
    <h2>Steps for grouping in the Muench method</h2>
    <p class="item">Define a <code>key</code> for the property we want
    to use for grouping.</p>
    <p class="item">Select all of the nodes ...</p>
    <p class="note">This can be really complicated.</p>
    <p class="note">Many people don't enjoy this method.</p>
    <p class="note">XSLT 2.0 attempts to make grouping simpler.</p>
```

```
    <p class="item">For each unique grouping value ...</p>
    <h2>Steps for grouping in XSLT 2.0</h2>
    <p class="item">Define an XPath expression ...</p>
    <p class="item">Select all of the nodes ...</p>
    <p class="note">This is much easier than it used to be.</p>
    <p class="item">Instead of dealing with each ...</p>
  </body>
</html>
```

We now have two classes of `<p>` elements. Those with `class="item"` will be processed just as before. The `class="note"` elements will be put in an ordered list. To add another layer of complexity to our example, we'll number all of the `class="note"` items sequentially throughout the document. In other words, if the first group of adjacent `class="note"` elements has three members, we want to start numbering the next group of `class="note"` elements at four. Here's the stylesheet, which we'll discuss in detail in just a minute:

```
<?xml version="1.0"?>
<!-- for-each-group_group-adjacent2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" include-content-type="no"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Grouping with group-adjacent</title>
      </head>
      <body style="font-family: sans-serif;">
        <h1>Grouping with group-adjacent</h1>
        <xsl:for-each-group select="html/body/*"
          group-adjacent="if (self::p[@class='item']) then 1
                          else if (self::p[@class='note']) then 2
                          else 3">
          <xsl:choose>

            <!-- group for <p class="item"> -->
            <xsl:when test="current-grouping-key() = 1">
              <ul>
                <xsl:for-each select="current-group()">
                  <li>
                    <xsl:copy-of select="@*[not(name()='class')]"/>
                    <xsl:apply-templates select="*|text()"/>
                  </li>
                </xsl:for-each>
              </ul>
            </xsl:when>

            <!-- group for <p class="note"> -->
            <xsl:when test="current-grouping-key() = 2">
              <xsl:variable name="starting-point">
                <xsl:number count="p[@class='note']"
                  level="any" format="1"/>
```

```
            </xsl:variable>
            <table border="0" cellpadding="5" width="40%">
              <tr>
                <td width="10%">
                  <p><xsl:text>&#x20;</xsl:text></p>
                </td>
                <td style="background: #CCCCCC;">
                  <p style="font-weight: bold;">
                    <xsl:value-of
                      select="if (count(current-group()) gt 1)
                              then 'Notes'
                              else 'Note'"/>
                  </p>
                  <ol start="{$starting-point}">
                    <xsl:for-each select="current-group()">
                      <li>
                        <xsl:copy-of select="@*[not(name()='class')]"/>
                        <xsl:apply-templates select="*|text()"/>
                      </li>
                    </xsl:for-each>
                  </ol>
                </td>
              </tr>
            </table>
          </xsl:when>

          <!-- group for everything else -->
          <xsl:otherwise>
            <xsl:for-each select="current-group()">
              <xsl:apply-templates select="."/>
            </xsl:for-each>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:for-each-group>
    </body>
  </html>
</xsl:template>

<xsl:template match="*">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

First of all, notice that our `group-adjacent` attribute is significantly more complicated. We're using integer values here instead of `true` and `false` because we have more than two groups. (Remember, the grouping key for `group-adjacent` can be any atomic value; it doesn't have to be `true` or `false`.) Here's the `<xsl:for-each-group>` element:

```
group-adjacent="if (self::p[@class='item']) then 1
                else if (self::p[@class='note']) then 2
                else 3"
```

So any <p class="item"> element has a grouping key of 1, any <p class="note"> element has a grouping key of 2, and everything else has a grouping key of 3. Adding an <xsl:message> to the stylesheet as we did before, we can list the grouping keys:

```
current-grouping-key() = 3

current-grouping-key() = 1
current-grouping-key() = 1

current-grouping-key() = 2
current-grouping-key() = 2
current-grouping-key() = 2

current-grouping-key() = 1
current-grouping-key() = 1

current-grouping-key() = 3

current-grouping-key() = 1
current-grouping-key() = 1

current-grouping-key() = 2

current-grouping-key() = 1
```

Aside from the fact that we have three groups instead of two, the main complication here is how we process items in the second group. For the first group of <p class="note"> elements, we could simply use an ordered list (<ol>) element to number the items in the group. For every other group, we have to determine where to start numbering. This means we have to find the position of the first <p class="note"> element in all of the <p class="note"> elements to see where a particular group starts. If a group begins with the 38th <p class="note"> element, we need to generate <ol start="38"> in the HTML output document. <xsl:number> is perfect for this task.

For each group of <p class="note"> elements, we have three tasks:

1. Generate a heading for the notes. If there is more than one item in the current group, the heading is <h3>Notes</h3>; otherwise, it is <h3>Note</h3>.
2. Generate the start attribute for the <ol> element. We use <xsl:number> to count all of the <p class="note"> elements and tell us the position of the first item in the group.
3. Process the item itself. This means putting the current paragraph into a list item (<li>) element and processing its attributes and children.

Our first step is to generate the heading text. Fortunately, this is pretty simple; we use XPath 2.0's if operator:

```
<p style="font-weight: bold;">
  <xsl:value-of
       select="if (count(current-group()) gt 1)
               then 'Notes'
```

```
                else 'Note'"/>
</p>
```

We also use the new XPath `gt` operator to compare the size of the current group—`count(current-group( ))`—to the number `1`. If the current group has more than one item, the heading text is `Notes`; otherwise, it is `Note`.

Now that our first step is complete, we'll create a variable with the value of the `<ol>` `start` attribute:

```
<xsl:variable name="starting-point">
  <xsl:number count="p[@class='note']" level="any" format="1"/>
</xsl:variable>
```

Notice that we're using `format="1"` to create a numeric value. From an XSLT point of view, we don't have to worry about the datatype of the value. (There's no need to add `as="xs:integer"` to the `<xsl:variable>` element.) The HTML renderer treats this value as a number and uses it accordingly.

The last step is simply to output the new `<ol>` element (using the `$starting-point` variable we just initialized) and process the `<p>` element as before:

```
<ol start="{$starting-point}">
  <xsl:for-each select="current-group()">
    <li>
      <xsl:copy-of select="@*[not(name()='class')]"/>
      <xsl:apply-templates select="*|text()"/>
    </li>
  </xsl:for-each>
</ol>
```

We copy all of the attributes of the HTML `<p>` element *except* the `class` attribute. Having copied the attributes, we use `<xsl:apply-templates>` to process anything that might be in the original paragraph. As in our earlier stylesheet, we used an attribute value template to include the `start` attribute on the `<ol>` element.

Our generated HTML document looks like Figure 7-2.

Grouping using group-starting-with

The next grouping type we'll use is `group-starting-with`. With this attribute, we'll specify the node or condition that starts a new group. From that point, every item goes into the new group until another starting node or condition is found. After that, the XSLT processor closes the current group and starts a new one.

We'll use this approach to add some structure to another HTML document. In this case, we want to create a group around every `<h1>` element. We'll use the file *group-adjacent_input.html* from our earlier example.

We'll transform this markup into DocBook, which uses a containment strategy instead of HTML's sequential approach. In other words, in HTML a level 1 heading (`<h1>`)

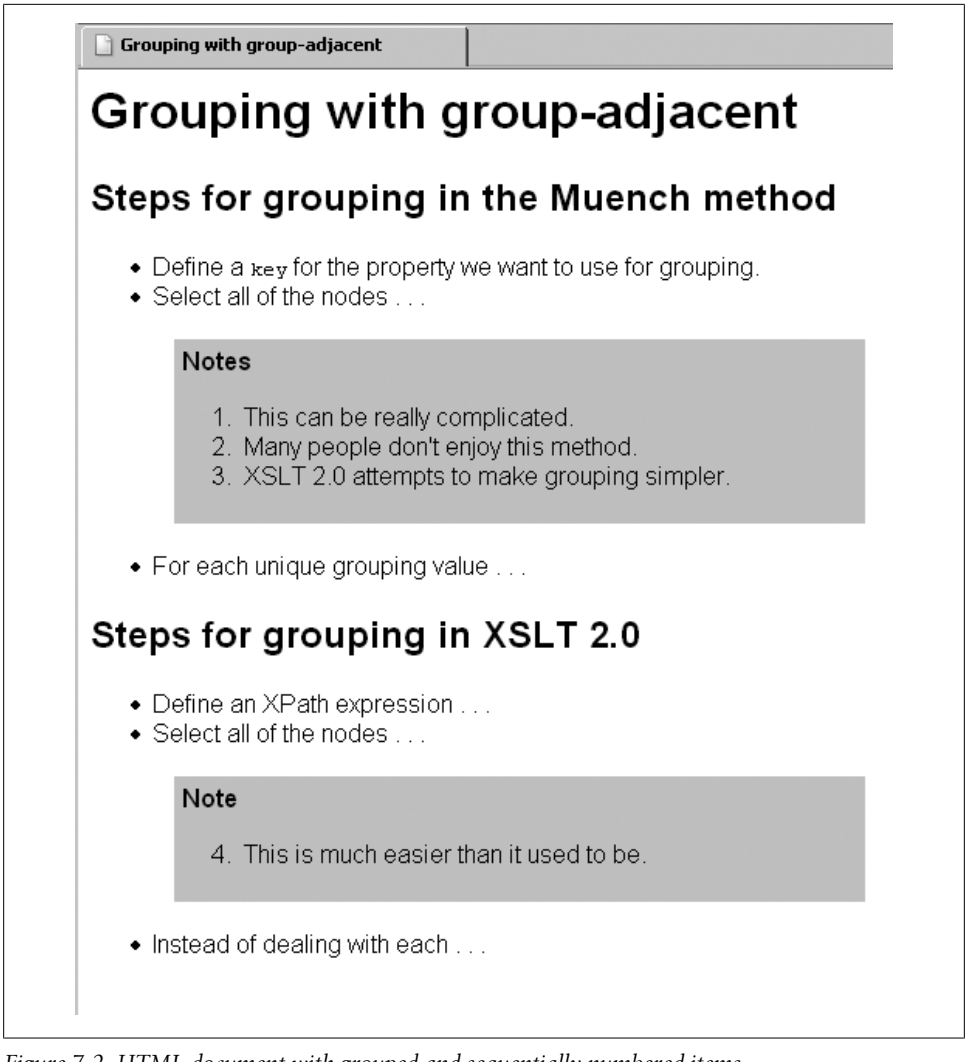

Grouping with group-adjacent
Grouping with group-adjacent
Steps for grouping in the Muench method
Define a key for the property we want to use for grouping.
Select all of the nodes . . .
Notes
1. This can be really complicated.
2. Many people don't enjoy this method.
3. XSLT 2.0 attempts to make grouping simpler.
For each unique grouping value . . .
Steps for grouping in XSLT 2.0
Define an XPath expression . . .
Select all of the nodes . . .
Note
4. This is much easier than it used to be.
Instead of dealing with each . . .

Figure 7-2. HTML document with grouped and sequentially numbered items

stands on its own. We can assume that the first `<h1>` starts a section and that the next `<h1>` ends this section, but there's no guarantee that an HTML document is structured that way.

In DocBook, a level 1 heading is part of a section, not a standalone element. The section starts with a `<sect1>` tag and contains everything through the `</sect1>` tag, including a `<title>` element that contains the text we'd normally put on the HTML `<h1>` element.

Here's how our stylesheet looks:

```
<?xml version="1.0"?>
<!-- for-each-group_group-starting-with.xsl -->
```

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/">
    <chapter>
      <title>Grouping in XSLT</title>
      <xsl:apply-templates select="html/body"/>
    </chapter>
  </xsl:template>

  <xsl:template match="body">
    <xsl:for-each-group select="*" group-starting-with="h1">
      <sect1>
        <xsl:apply-templates select="current-group()"/>
      </sect1>
    </xsl:for-each-group>
  </xsl:template>

  <xsl:template match="h1">
    <title>
      <xsl:apply-templates/>
    </title>
  </xsl:template>

  <xsl:template match="p">
    <para>
      <xsl:apply-templates/>
    </para>
  </xsl:template>

  <xsl:template match="*">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

Our well-structured results look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<chapter>
   <title>Grouping in XSLT</title>
   <sect1>
      <h2>Steps for grouping in the Muench method</h2>
      <para>Define a <code>key</code> for the property we want
    to use for grouping.</para>
      <para>Select all of the nodes ...</para>
      <para>For each unique grouping value, ...</para>
      <h2>Steps for grouping in XSLT 2.0</h2>
      <para>Define an XPath expression ...</para>
      <para>Select all of the nodes we want to group ...</para>
      <para>Instead of dealing with each ...</para>
```

```
    </sect1>
</chapter>
```

In the generated DocBook code, each `<h1>` becomes the start of a `<sect1>` element. The text of the `<h1>` becomes the `<title>` of the section, each HTML `<p>` element becomes a DocBook `<para>` element, and everything else is copied as is to the output.

Grouping Using group-ending-with

The last grouping style is `group-ending-with`. This is similar to `group-starting-with`, only we're specifying the condition that *ends* each group. We'll use this technique to insert items into a three-column HTML table. Each row in the table is created from a group; each group will have no more than three members. We handle the special case of a row with less than three members (`count(current-group( )) lt 3`) by using a simplified version of our list of cars:

```
<?xml version="1.0"?>
<!-- carlist.xml -->
<cars>
  <make>Alfa Romeo</make>
  <make>Bentley</make>
  <make>Chevrolet</make>
  <make>Dodge</make>
  <make>Eagle</make>
  <make>Ford</make>
  <make>GMC</make>
  <make>Honda</make>
  <make>Isuzu</make>
  <model>Javelin</model>
  <model>K-Car</model>
  <make>Lincoln</make>
  <make>Mercedes</make>
  <make>Nash</make>
  <make>Opel</make>
  <make>Pontiac</make>
  <model>Quantum</model>
  <model>Rambler</model>
  <make>Studebaker</make>
</cars>
```

There are 19 cars in our list, so the final group should have only one item in it. Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- for-each-group_group-ending-with.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" include-content-type="no"/>

  <xsl:template match="/">
    <html>
      <head>
```

```
        <title>Car Makes and Models</title>
      </head>
      <body style="font-family: sans-serif;">
        <h1>Car Makes and Models</h1>
        <p>Here are the car makes and models in
        our input document.</p>
        <table border="1" cellpadding="5">
          <xsl:apply-templates select="cars"/>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="cars">
    <xsl:for-each-group select="make|model"
      group-ending-with="*[position() mod 3 = 0]">
      <tr>
        <xsl:apply-templates select="current-group()"/>
        <xsl:if test="count(current-group()) lt 3">
          <td style="background: #CCCCCC;"
            colspan="{3 - count(current-group())}">
          </td>
        </xsl:if>
      </tr>
    </xsl:for-each-group>
  </xsl:template>

  <xsl:template match="make">
    <td style="font-weight: bold;">
      <xsl:apply-templates/>
    </td>
  </xsl:template>

  <xsl:template match="model">
    <td style="font-style: italic; font-weight: bold;">
      <xsl:apply-templates/>
    </td>
  </xsl:template>

</xsl:stylesheet>
```

Figure 7-3 displays our results.

The grouping key depends on the position of each element. In our input document, all of the items we're grouping (the `<make>` and `<model>` elements) are at the same level of the document, so the `position( )` function tells us what we need to know. The `position( )` returns the position of an item within a sequence. In other words, the element `<xsl:for-each-group select="make|model" group-by="*[position( ) mod 3 = 0]">` creates a sequence containing all `<make>` and `<model>` elements. Calling `position( )` returns an item's position in that sequence.

That covers all of the new grouping options in XSLT 2.0. As we said before, all of the things we did here can be done in XSLT 1.0, but the XSLT 2.0 syntax makes it much easier to create and maintain stylesheets that do complicated grouping.

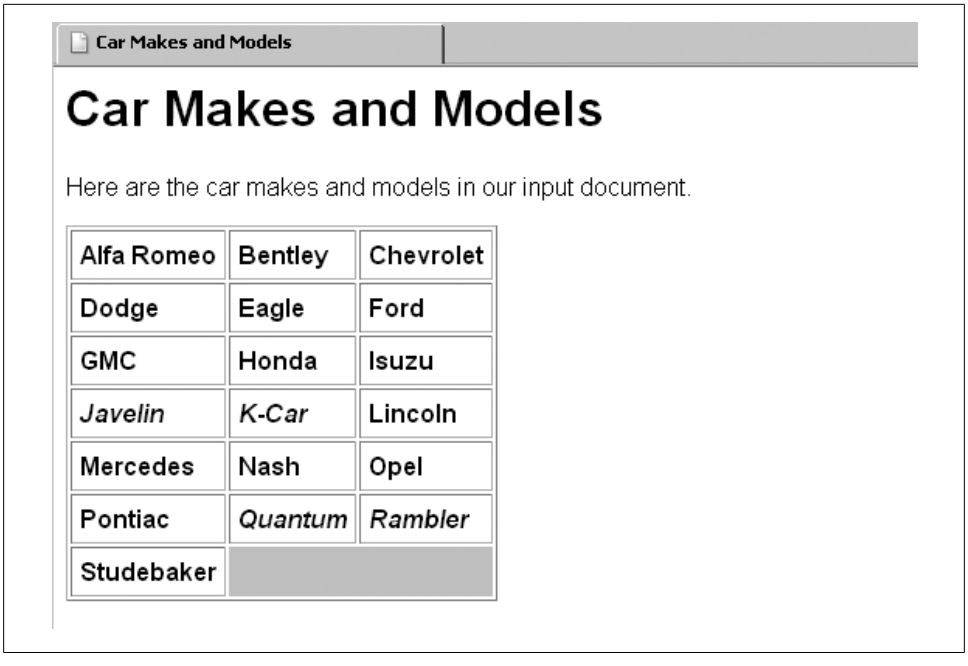


Figure 7-3. HTML document with items grouped into columns

Summary

In this chapter, we've gone over all of the common techniques used for sorting and grouping elements, including the new features available in XSLT 2.0. Regardless of the kinds of stylesheets you'll need to write in your XML projects, you'll probably use these techniques in everything you do. Now that we've covered how to sort and group elements, the next chapter will talk about how to work with multiple documents; that topic builds on what we've covered here.

CHAPTER 8

Combining Documents

One of XSLT's most powerful features is the `document( )` function, which lets you combine documents. You can use parts of a document (specified with XPath expressions, of course) to identify other documents. You can then open those documents and perform stylesheet functions on the combination of those documents. In this chapter, we'll cover the `document( )` function in all its glory.

[2.0] XSLT 2.0 and XPath 2.0 provide three new functions for combining documents: `doc( )`, `collection( )`, and `unparsed-text( )`. The `doc( )` function, defined by XPath 2.0, is similar to the `document( )` function, though less powerful. The `collection( )` function, also defined by XPath 2.0, allows us to add collections of nodes provided by an XSLT processor. The kinds of nodes provided can vary from one processor to the next. Finally, the `unparsed-text( )` function lets us add raw text to the data we're processing in our stylesheet. When combined with features such as tokenization and regular expressions, `unparsed-text( )` gives us many new ways of manipulating data. We'll look at these new functions at the end of this chapter.

The document( ) Function

A common task in writing stylesheets is combining data from different documents. We'll start by using the `document( )` function to parse and process multiple XML documents. We'll start our discussion with XML-tagged purchase orders that look like this:

```
<?xml version="1.0"?>
<!-- po38295.xml -->
<purchase-order id="38295">
  <date year="2001" month="9" day="8"/>
  <customer id="4738" level="Basic">
    <address type="business">
      <name>
        <title>Ms.</title>
        <first-name>Amanda</first-name>
        <last-name>Reckonwith</last-name>
      </name>
      <street>930-A Chestnut Street</street>
```

```
      <city>Lynn</city>
      <state>MA</state>
      <zip>02930</zip>
    </address>
    <address type="ship-to"/>
  </customer>
  <items>
    <item part-no="23813-03-CDK">
      <name>Cucumber Decorating Kit</name>
      <qty>1</qty>
      <price>29.95</price>
    </item>
  </items>
</purchase-order>
```

If we had a few dozen documents like this, we might want to view the collection of purchase orders in a number of ways. We could view them sorted (or even grouped) by customer, by part number, by the amount of the total order, by the state to which they were shipped, etc. One way to do this would be to write code that worked directly with the Document Object Model. We could parse each document, retrieve its DOM tree, and then use DOM functions to order and group the various DOM trees, display certain parts of the DOM trees, etc. Because this is an XSLT book, though, you probably won't be surprised to learn that XSLT provides a function to handle most of the heavy lifting for us.

We'll start with a simple example that uses the `document( )` function. We'll assume that we have several purchase orders and that we want to combine them into a single report document. One thing we can do is create a *master document* that references all the purchase orders we want to include in the report. Here's what that master document might look like:

```
<?xml version="1.0"?>
<!-- polist.xml -->
<report>
  <title>Selected Purchase Orders</title>
  <po filename="po38292.xml"/>
  <po filename="po38293.xml"/>
  <po filename="po38294.xml"/>
  <po filename="po38295.xml"/>
</report>
```

We'll fill in the details of our stylesheet as we go along, but the first several stylesheets we'll create will use the `document( )` function to open multiple input files. Once we've opened those files, we can use the templates in our stylesheet to transform their contents.

For our first stylesheet, we'll use the `filename` attribute as the argument to the `document( )` function. The simplest thing we can do is open each purchase order, then write its details to the output stream. The key to our stylesheet is this call to the `document( )` function:

Figure 8-1. Document generated from multiple input files

```
<xsl:for-each select="/report/po">
  <xsl:apply-templates
    select="document(@filename)/purchase-order"/>
</xsl:for-each>
```

When we process our master document with this stylesheet, the results look like Figure 8-1.

The most notable thing about our results is that we've been able to generate a document that contains the contents of several other documents. To keep our example short, we've only combined four purchase orders, but there's no limit (beyond the physical limits of our machine) to the number of documents we could combine. Best of all, we didn't have to modify any of the individual purchase orders to generate our report.

Here's the complete stylesheet:

```
<?xml version="1.0"?>
<!-- masterdox1.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title><xsl:value-of select="/report/title"/></title>
      </head>
      <body style="font-family: sans-serif;">
```

```
      <h1>Selected Purchase Orders - Unsorted</h1>
      <xsl:for-each select="/report/po">
        <xsl:apply-templates
          select="document(@filename)/purchase-order"/>
      </xsl:for-each>
    </body>
  </html>
</xsl:template>

<xsl:template match="purchase-order">
  <h2>
    <xsl:value-of
      select="customer/address[@type='business']/name/title"/>
    <xsl:text> </xsl:text>
    <xsl:value-of
      select="customer/address[@type='business']/name/first-name"/>
    <xsl:text> </xsl:text>
    <xsl:value-of
      select="customer/address[@type='business']/name/last-name"/>
    <xsl:text> - </xsl:text>
    <xsl:value-of
      select="customer/address[@type='business']/city"/>
    <xsl:text>, </xsl:text>
    <span style="font-weight: bold;">
      <xsl:value-of
        select="customer/address[@type='business']/state"/>
    </span>
  </h2>
  <p>
    <xsl:text>Ordered on </xsl:text>
    <xsl:value-of select="date/@month"/>
    <xsl:text>/</xsl:text>
    <xsl:value-of select="date/@day"/>
    <xsl:text>/</xsl:text>
    <xsl:value-of select="date/@year"/>
    <xsl:text>:</xsl:text>
  </p>
  <table width="80%" border="0">
    <tr style="background: #66FF66;">
      <th>Item</th>
      <th>Quantity</th>
      <th>Price Each</th>
      <th>Total</th>
    </tr>
    <xsl:for-each select="items/item">
      <tr>
        <xsl:attribute name="style">
          <xsl:text>background: </xsl:text>
          <xsl:choose>
            <xsl:when test="position() mod 2">
              <xsl:text>#CCCCFF</xsl:text>
            </xsl:when>
            <xsl:otherwise>
              <xsl:text>#66FF66</xsl:text>
            </xsl:otherwise>
```

```
          </xsl:choose>
          <xsl:text>;</xsl:text>
        </xsl:attribute>
        <td width="40%">
          <span style="font-weight: bold;">
            <xsl:value-of select="name"/>
          </span>
          <xsl:text> (part #</xsl:text>
          <xsl:value-of select="@part-no"/>
          <xsl:text>)</xsl:text>
        </td>
        <td style="text-align: center;" width="20%">
          <xsl:value-of select="qty"/>
        </td>
        <td style="text-align: right;" width="20%">
          <xsl:value-of select="price"/>
        </td>
        <td style="text-align: right;" width="20%">
          <xsl:choose>
            <xsl:when test="position()=1">
              <xsl:value-of
                select="format-number(price * qty, '$#,###.00')"/>
            </xsl:when>
            <xsl:otherwise>
              <xsl:value-of
                select="format-number(price * qty, '#,###.00')"/>
            </xsl:otherwise>
          </xsl:choose>
        </td>
      </tr>
    </xsl:for-each>
    <tr style="font-weight: bold;">
      <td colspan="3" style="text-align: right;">
        Total:
      </td>
      <td style="text-align: right; color: white; background: black;">
        <xsl:variable name="orderTotal">
          <xsl:call-template name="sumItems">
            <xsl:with-param name="items" select="items/item" />
          </xsl:call-template>
        </xsl:variable>
        <xsl:value-of
          select="format-number($orderTotal, '$#,###.00')"/>
      </td>
    </tr>
  </table>
</xsl:template>

<xsl:template name="sumItems">
  <xsl:param name="items" />
  <xsl:param name="runningTotal" select="0" />
  <xsl:choose>
    <xsl:when test="$items">
      <xsl:variable name="firstItemSubtotal"
        select="$items[1]/qty * $items[1]/price" />
```

```
        <xsl:call-template name="sumItems">
          <xsl:with-param name="items" select="$items[position() > 1]" />
          <xsl:with-param name="runningTotal"
            select="$runningTotal + $firstItemSubtotal" />
        </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="$runningTotal" />
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

</xsl:stylesheet>
```

An Aside: Doing Math with Recursion

While we're here, we'll also mention the recursive technique we used to calculate the total for each purchase order. At first glance, this seems like a perfect opportunity to use the `sum( )` function. We want to add the total of the price of each item multiplied by its quantity. We could try to invoke the `sum( )` function like this:

```
<xsl:value-of select="sum(item/qty*item/price)"/>
```

Unfortunately, the `sum( )` function simply takes the node-set passed to it, converts each item in the node-set to a number, and then returns the sum of all of those numbers. The expression `item/qty*item/price`, while a perfectly valid XPath expression, isn't a valid node-set. With that in mind, we have to create a recursive `<xsl:template>` to do the work for us. There are a couple of techniques worth mentioning here; we'll go through them in the order we used them in our stylesheet.

[2.0] If you're using an XSLT 2.0 processor, you can use new functions of XPath 2.0 to avoid recursion altogether. XSLT 2.0 makes this stylesheet much simpler and shorter; we discuss all the details in the last section of this chapter. If you're using an XSLT 2.0 processor and you don't really care how recursion works, feel free to skip ahead.

Recursive design

First, we pass the set of all `<items>` elements to our recursive template:

```
<xsl:variable name="orderTotal">
  <xsl:call-template name="sumItems">
    <xsl:with-param name="items" select="items/item" />
  </xsl:call-template>
</xsl:variable>
```

Our recursive template is named `sumItems`; the value it returns becomes the value of the variable `$orderTotal`. Within the recursive template, we use two parameters. One is the variable containing all of the `<item>` elements; the other is the total of all items we've processed so far. The first time we call the template, of course, this value is zero. (We gave the parameter a default value so we don't have to worry about it when we call the template for the first time.) Here are the two parameters to our template:

```
<xsl:template name="sumItems">
  <xsl:param name="items" />
  <xsl:param name="runningTotal" select="0" />
```

The body of our recursive template is an `<xsl:choose>` element. If the `$items` variable contains at least one node, we calculate the total for the current item (the price of the item multiplied by the quantity):

```
<xsl:when test="$items">
  <xsl:variable name="firstItemSubtotal"
    select="$items[1]/qty * $items[1]/price" />
```

Thinking back to how variables are converted to boolean values, a node-set (or sequence, if we're in XSLT 2.0) that starts with at least one node is `true`. Notice also that we're calculating the subtotal for the first item (`$items[1]`) only. Now that we've calculated the value of the current item, our template calls itself:

```
<xsl:call-template name="sumItems">
  <xsl:with-param name="items" select="$items[position() > 1]" />
  <xsl:with-param name="runningTotal"
    select="$runningTotal + $firstItemSubtotal" />
</xsl:call-template>
```

Notice that the `$items` parameter passed to the next invocation of the template contains everything after the first item (`position( ) > 1`). The parameter `$runningTotal` is the previous value of the parameter plus the total for the current item.

The template calls itself until eventually the parameter `$items` is empty. That means the attribute `test="$items"` returns `false`, so the `<xsl:otherwise>` branch of the `<xsl:choose>` element is taken. That branch uses `<xsl:value-of>` to output the final value. (All other invocations of the `sumItems` template don't output anything.)

Here's the complete recursive template:

```
<xsl:template name="sumItems">
  <xsl:param name="items" />
  <xsl:param name="runningTotal" select="0" />
  <xsl:choose>
    <xsl:when test="$items">
      <xsl:variable name="firstItemSubtotal"
        select="$items[1]/qty * $items[1]/price" />
      <xsl:call-template name="sumItems">
        <xsl:with-param name="items" select="$items[position() > 1]" />
        <xsl:with-param name="runningTotal"
          select="$runningTotal + $firstItemSubtotal" />
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="$runningTotal" />
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Recursion is a common technique in XSLT 1.0 stylesheets. XSLT 2.0 eliminates the need for recursion in many cases (more on that in a minute), but recursion is sometimes

the only way to solve problems in XSLT. Knowing how to use this technique will serve you well.

Using format-number( ) to control output

The final nicety in our stylesheet is that we use the XSLT `format-number( )` function to display the total for the current purchase order. We've already discussed how we set the value of the variable `$orderTotal` to be the output of the template named `sumItems`; once the variable is set, we use `format-number` to display it with a currency sign, commas, and two decimal places:

```
<xsl:value-of select="format-number($order-total, '$#,###.00')"/>
```

Base URIs and the document( ) Function

In our previous stylesheet, we used the `document( )` function to select some number of nodes from the original source document (our list of purchase orders), and then open those files. There are a number of ways to invoke the `document( )` function; we'll discuss them briefly here.

The most common way to use the `document( )` function is as we just did. We use an XPath expression to describe a node-set; the `document( )` function takes each node in the node-set, converts it to a string, and then uses that string as a URI. So, when we passed a node-set containing the `filename` attributes in the list of purchase orders, each one is used as a URI. If those URIs are relative references (i.e., they don't begin with a protocol such as `http`), the XSLT processor needs a base URI. If the argument is a node, the base URI of the node is used to resolve the relative reference. If the argument is a string, as it is here, the base URI of the stylesheet is used.

Every node in the XPath source tree is associated with a *base URI*. When using the `document( )` function, the base URI is important for resolving references to various resources specified with relative links.

Here I'll offer more detail about base URIs than you're ever likely to need: If a given node is an element or processing instruction node, and that node occurs in an external entity, then the base URI for that node is the base URI of the external entity. If an element or processing instruction node does not occur in an external entity, then its base URI is the base URI of the document in which it appears. The base URI of a document node is the base URI of the document itself, and the base URI of an attribute, comment, namespace, or text node is the base URI of that node's parent.

To set the base URI of a node in an XML source document, use the `xml:base` attribute. If a given node doesn't have a base URI, the XSLT processor looks at the node's ancestors (the node's parent, then the node's parent's parent, and so forth) until it finds an `xml:base` attribute. If neither a node nor its ancestors have a `xml:base` attribute, the base URI of the stylesheet is used. See the definition of the `base-uri( )` function for more information on base URIs.

If the `document( )` function has two arguments, the second must be a node-set. The first argument is processed as just described. The second argument is used to set the base URI. The base URI of the first node in the node-set (or sequence, in XSLT 2.0) is the base URI. The two-argument form of the `document( )` function isn't used often, but it's there if you need it.

In our example, we're using strings without any particular protocol (we're using `po38293.xml` instead of `http://.../po38293.xml`). You can also pass a URL to the `document( )` function. If we wanted to open a particular resource, we could simply pass the name of the resource:

```
document('http://www.ibm.com/developerworks/news/dw_dwtp.rss')
```

This action would open this particular resource and process it. Be aware that XSLT processors are required to either signal an error and halt *or* return an empty node-set if a resource can't be found. XSLT processors also don't have to support any particular protocols (`http`, `ftp`, etc.); you have to check the documentation of your XSLT processor to see what protocols are supported.

Finally, a special case occurs when you pass an empty string to the `document( )` function. As we've discussed the various combinations of arguments that can be passed to the function, we've gone over the rules for resolving URIs. When we call `document('')`, the XSLT processor parses the current stylesheet and returns a single node—the root node of the stylesheet itself. This technique is very useful for processing lookup tables in a stylesheet, which we'll discuss later in this chapter.

The document( ) Function and Sorting

Up to now, we've written a simple XML document that contains references to other XML documents, then we created a stylesheet that combines all those referenced XML documents into a single output document. That's all well and good, but we'll probably want to do more advanced things. For example, it might be useful to generate a document that lists all items ordered across all purchase orders. It might also be useful to sort all the purchase orders by the state to which they were shipped or by the last name of the customer. We'll go through some of these scenarios to illustrate the design challenges we face when generating documents from multiple input files.

Our first challenge will be to generate a listing of all purchase orders and sort them by state, then by city within state. This isn't terribly difficult; we'll simply use the `<xsl:sort>` element in conjunction with the `document( )` function. Here's the heart of our new stylesheet:

```
<?xml version="1.0"?>
<!-- masterdox2.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
...
<xsl:apply-templates
  select="document(/report/po/@filename)/purchase-order">
  <xsl:sort select="customer/address/state"/>
  <xsl:sort select="customer/address/city"/>
</xsl:apply-templates>
```

Here we're selecting all of the `<purchase-order>` elements and sorting them by the values of their `<state>` and `<city>` elements. Figure 8-2 shows our output document, sorted by the value of the `<state>` element (the state abbreviation) in each purchase order.

Notice that we're sorting purchase orders by the state *abbreviation*, not the actual state name; we'll address that in our next example.

Implementing Lookup Tables

We mentioned earlier that calling the `document( )` function with an empty string enabled us to access the nodes in the stylesheet itself. We can use this behavior to implement a lookup table. As an example, we'll create a lookup table that associates an abbreviation such as `ME` with the state name `Maine`. We can then use the value from the lookup table as the sort key. More attentive readers might have noticed in our previous example that although the abbreviation `MA` does indeed sort before the abbreviation `ME`, a sorted list of the state names themselves would put `Maine` (abbreviation `ME`) before `Massachusetts` (abbreviation `MA`).

First, we'll create our lookup table. We'll use the fact that a stylesheet can have any element as a top-level element, provided that element is namespace-qualified to

Figure 8-2. Purchase orders sorted by state abbreviation

distinguish it from the `xsl:` namespace reserved for stylesheets. Here's the namespace prefix definition and part of the lookup table that uses it:

```
<?xml version="1.0"?>
<!-- masterdox3.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:states="http://www.usps.com/ncsc/lookups/abbreviations.html"
  exclude-result-prefixes="states">

  <states:name abbrev="AL">Alabama</states:name>
  <states:name abbrev="AK">Alaska</states:name>
  <states:name abbrev="AS">American Samoa</states:name>
  <!-- Many state names deleted for brevity -->
  <states:name abbrev="WV">West Virginia</states:name>
  <states:name abbrev="WI">Wisconsin</states:name>
  <states:name abbrev="WY">Wyoming</states:name>
```

(The namespace mapped to the `states` prefix is the URL for the official list of state abbreviations from the United States Postal Service, although it could be any string.)

To look up values in our table, we'll use the `document( )` function to return the root node of our stylesheet, then we'll look for a `<states:name>` element with a `abbrev` attribute that matches the value of the current `<state>` element in the purchase order we're currently processing. Here's the somewhat convoluted syntax that performs this magic:

```
<body style="font-family: sans-serif;">
  <h1>
```

```
        Selected Purchase Orders - Sorted by state
        <span style="font-style: italic;
                     text-decoration: underline;">name</span>
      </h1>
      <xsl:for-each
        select="document(/report/po/@filename)/purchase-order
                /customer/address/state">
        <xsl:sort
          select="document('')/*/states:name[@abbrev=current()]"/>
        <xsl:sort select="../city"/>
        <xsl:apply-templates select="ancestor::purchase-order"/>
      </xsl:for-each>
    </body>
```

Notice that we use the `document( )` function twice; once to open the document referred to by the `filename` attribute, and once to open the stylesheet itself. We also need to discuss the XPath expression in the `select` attribute of the `<xsl:sort>` element. There are four significant parts to this expression:

`document('')`
: Returns the root node of the current stylesheet.

`/*/`
: Indicates that what follows must be a top-level element of the stylesheet. This syntax starts at the root of the document and has a single element. For our current stylesheet, we could have written the XPath expression like this:

```
select="document('')/xsl:stylesheet/
        states:name[@abbrev=current()]"
```

  Because the root element of a stylesheet can be either `<xsl:stylesheet>` or `<xsl:transform>`, it's better to use the asterisk.

`states:name`
: We're looking for a `name` element in the `states:` namespace.

`[@abbrev=current( )]`
: Means that the `abbrev` attribute of the current `<states:name>` element has the same value as the current node. We have to use the XSLT `current( )` function here because we want the current node, not the context node. Inside the predicate expression, the current node is the `<state>` element that we're processing, whereas the context node is the `<states:name>` element that contains the `abbrev` attribute that we evaluate.

Notice that the `<xsl:stylesheet>` element contains the `exclude-result-prefixes` attribute to ensure the output document doesn't declare the `states:` prefix. We use only elements in this namespace to implement the lookup table; we certainly don't need to have the `states:` namespace defined in the HTML output document.

As with any stylesheet, we can write our XPath expressions in different manners. Here is one way to access the lookup table:

```
<xsl:for-each
  select="document(/report/po/@filename)/purchase-order
          /customer/address/state">
  <xsl:sort
    select="document('')/*/states:name[@abbrev=current()]"/>
  <xsl:sort select="../city"/>
  <xsl:apply-templates select="ancestor::purchase-order"/>
</xsl:for-each>
```

And here's another:

```
<xsl:for-each
  select="document(/report/po/@filename)/purchase-order">
  <xsl:sort
    select="document('')/*/states:name
            [@abbrev=current()/customer/address/state]"/>
  <xsl:sort select="customer/address/city"/>
  <xsl:apply-templates select="."/>
</xsl:for-each>
```

Both of these listings do the same thing, but they specify parts of the document in different ways. The first example creates a `<xsl:for-each>` for all of the `<state>` elements; the second uses the `<purchase-order>` elements. In each case, the remaining XPath expressions are written from a different position in the node tree. Given a `<state>` element, the related `<city>` element is at `../city`, and we have to use the `ancestor` axis to process the `<purchase-order>`. If we base our `<xsl:for-each>` on the `<purchase-order>` element, we specify the `<city>` element with `customer/address/city`. In the second example, specifying the `<purchase-order>` is done with a dot.

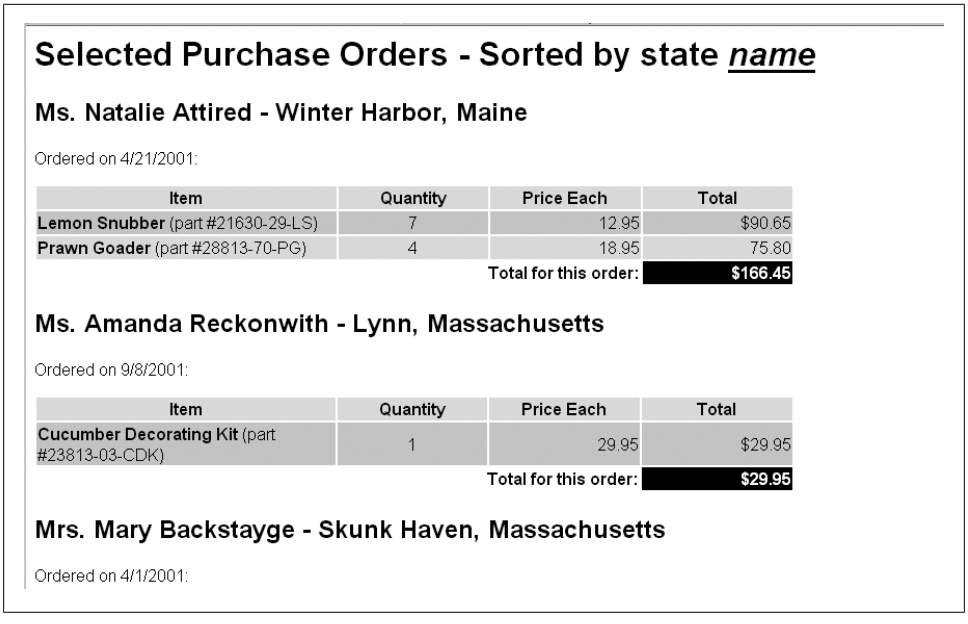
Both of the samples are similarly complex, but you should always be on the lookout for simpler ways to write your XPath expressions. If every XPath expression inside an `<xsl:for-each>` or `<xsl:template>` element is very complex, you might be able to change the outermost expression and simplify your stylesheet.

Figure 8-3 shows the output from the stylesheet with a lookup table.

Notice that now the purchase orders have been sorted by the actual name of the state referenced in the address, not by the state's abbreviation. Lookup tables are an extremely useful side effect of the way the `document('')` function works. You could place a lookup table in another file and use the `document( )` function to load that information from the other file, but the technique we've employed here is the most common way to implement lookup tables.

Grouping Across Multiple Documents

Our final task will be to group our collection of purchase orders. We'll create a new listing that groups all the purchase orders by the state to which they were shipped. We'll use the grouping technique we used earlier in Chapter 7.



Selected Purchase Orders - Sorted by state *name*

Ms. Natalie Attired - Winter Harbor, Maine

Ordered on 4/21/2001:

Item	Quantity	Price Each	Total
Lemon Snubber (part #21630-29-LS)	7	12.95	$90.65
Prawn Goader (part #28813-70-PG)	4	18.95	75.80
		Total for this order:	**$166.45**

Ms. Amanda Reckonwith - Lynn, Massachusetts

Ordered on 9/8/2001:

Item	Quantity	Price Each	Total
Cucumber Decorating Kit (part #23813-03-CDK)	1	29.95	$29.95
		Total for this order:	**$29.95**

Mrs. Mary Backstayge - Skunk Haven, Massachusetts

Ordered on 4/1/2001:

Figure 8-3. Purchase orders sorted by state name

We'll define an XSLT `key( )` for all of the `<state>` values in our set of purchase orders. We'll also create a variable that stores all of these orders. By reading all of the purchase orders into a variable, we won't have to call the `document( )` function every time we need to access a particular purchase order. With those two structures in place, we'll be ready to do grouping just as we did in the last chapter.

To get things started, here's how we define the `key( )`:

```
<xsl:key name="po-key" match="purchase-order"
  use="customer/address/state"/>
```

And here's how we load all of the purchase order documents into a variable:

```
<xsl:variable name="purchase-orders"
  select="document(/report/po/@filename)/purchase-order"/>
```

Now that we have a variable that contains all of the nodes from all of the purchase orders, we're ready to process them. We'll go through these steps:

1. Start an `<xsl:for-each>` element to process all of the nodes in the variable `$purchase-orders`.
2. Sort the purchase orders by state name. As with our previous stylesheet, this means using the `document('')` function to retrieve the state name that matches the abbreviation in the purchase order.
3. Save the value of the current `<state>` in a variable.

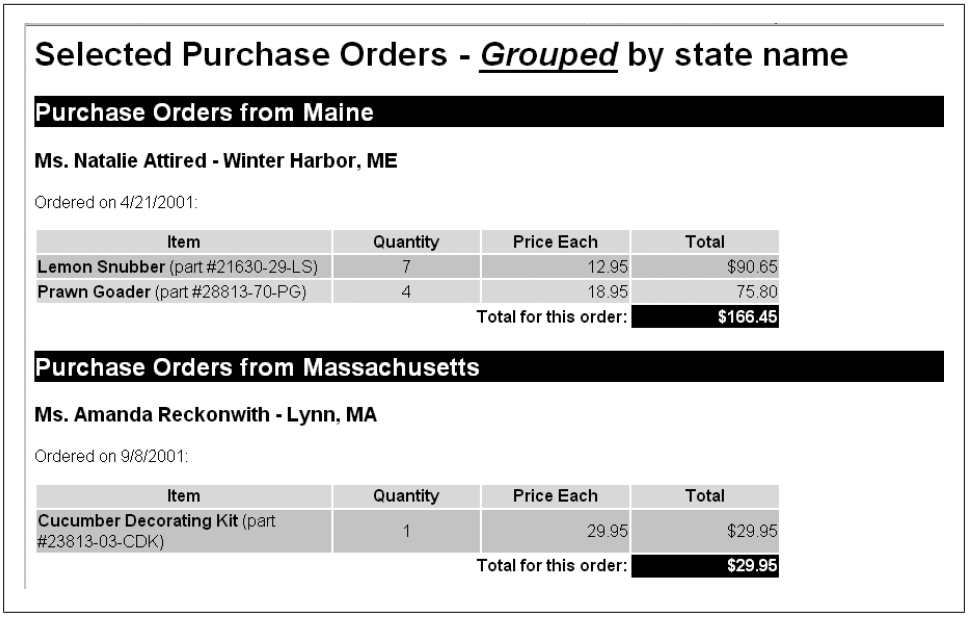
4. Use the key( ) function to save into a variable all of the purchase orders from the current state.
5. If this is the first time we've seen this particular state (we'll use generate-id( ) here), use <xsl:apply-templates> to process all the purchase orders that match the current state. When we call <xsl:apply-templates>, we'll sort all the purchase orders for that state by city before we process them.

Here's the significant portion of the stylesheet; the template for processing the <purchase-order> element is unchanged from our previous stylesheets:

```
<?xml version="1.0"?>
<!-- masterdox4.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:states="http://www.usps.com/ncsc/lookups/abbreviations.html"
  exclude-result-prefixes="states">

...

<xsl:for-each select="$purchase-orders">
  <xsl:sort
    select="document('')/*/states:name
            [@abbrev=current()/customer/address/state]"/>
  <xsl:variable name="currentState"
    select="customer/address/state"/>
  <xsl:variable name="currentStatePOs"
    select="$purchase-orders/key('po-key', $currentState)"/>
  <xsl:if
    test="generate-id() = generate-id($currentStatePOs[1])">
    <h2 style="color: white; background: black;">
      Purchase Orders from
      <xsl:value-of
        select="document('')/*/states:name
                [@abbrev=$currentState]"/>
    </h2>
    <xsl:apply-templates select="$currentStatePOs">
      <xsl:sort select="customer/address/city"/>
    </xsl:apply-templates>
  </xsl:if>
</xsl:for-each>
```

Notice that we used the key( ) function to retrieve all of the <purchase-order>s that are from the current state. All of those orders are stored in a variable. To determine whether we're seeing a purchase order for the first time, we generate an ID for the current order (generate-id( )) and compare that to the generated ID for the first item in the set of purchase orders for the current state (generate-id($currentStatePOs[1])). If the two generated IDs are the same, we know that we're seeing a state name for the first time.

The result of our hard work looks like Figure 8-4.



Selected Purchase Orders - *Grouped* by state name

Purchase Orders from Maine

Ms. Natalie Attired - Winter Harbor, ME

Ordered on 4/21/2001:

Item	Quantity	Price Each	Total
Lemon Snubber (part #21630-29-LS)	7	12.95	$90.65
Prawn Goader (part #28813-70-PG)	4	18.95	75.80
		Total for this order:	**$166.45**

Purchase Orders from Massachusetts

Ms. Amanda Reckonwith - Lynn, MA

Ordered on 9/8/2001:

Item	Quantity	Price Each	Total
Cucumber Decorating Kit (part #23813-03-CDK)	1	29.95	$29.95
		Total for this order:	**$29.95**

Figure 8-4. Purchase orders grouped by state name

[2.0] Using XSLT 2.0 to Simplify Things

We've mentioned several times in this chapter that XSLT 2.0 has capabilities that can greatly simplify our stylesheet. We'll look at those techniques in this section.

The XSLT 1.0 stylesheet we've developed here has several areas ripe for improvement:

- We have to use the XSLT 1.0 grouping technique. We can avoid this clumsiness by using XSLT 2.0's `<xsl:for-each-group>` to do the grouping for us. The XSLT processor does the work of finding all of the unique state values, simplifying our lives significantly.
- We have to use recursion to calculate the total of each purchase order. We can use new features of XPath 2.0 to calculate the total in one simple expression.
- Although the `document('')` technique we used works, it's a convoluted way of doing things. We can replace it with an XSLT function that takes a state abbreviation as input and returns the full name of that state. This gives us a much simpler syntax. Calling `getStateName()` is much easier to understand and maintain than `document('')/*/states:name[@abbrev=$next-state]`.
- It's a minor point, but we have to use an `<xsl:choose>` element in a couple of places. We can use the `if` operator defined in the XPath 2.0 and XQuery 1.0 Functions and Operators spec to simplify the stylesheet. Instead of `<xsl:choose>`, `<xsl:when>`, and `<xsl:otherwise>`, we'll use `if`, `then`, and `else`.

- Another minor point is that the date of each purchase order is stored in the XML source as a set of three attributes. We can use XSLT 2.0's support for XML Schema datatypes to create an `xs:date`, and then use the power of the `format-date( )` function to format the date more elegantly.

With these things in mind, we'll start simplifying our stylesheet.

Grouping by Distinct Values

One problem in our final XSLT 1.0 stylesheet is that we have to find all the distinct state values in all the purchase orders. We use the Muench method to do this, but the code is more difficult to write and maintain. To simplify things, we can use `<xsl:for-each-group>` to do everything at once. When using the `group-by` attribute, `<xsl:for-each-group>` automatically finds all of the unique values of the `<state>` element. Here's a fragment of the code:

```
<xsl:for-each-group
  select="document(/report/po/@filename)/purchase-order"
  group-by="customer/address/state">
  <xsl:sort
    select="states:getStateName(current-grouping-key())"/>
```

The `<xsl:for-each-group>` element groups all of the `<purchase-order>` elements by the value of their `<state>` elements. The `<xsl:sort>` sorts the groups by the actual state names: `current-grouping-key( )` returns the value of the current `<state>` element, and `states:getStateName( )` is an XSLT function (we'll discuss this in the later section, "Implementing Lookup Tables with <xsl:function>").

Comparing this stylesheet to our final XSLT 1.0 stylesheet, we've used these two elements to replace roughly 15 lines of code. Even better, the new code is much simpler and easier to maintain.

Because `<xsl:for-each-group>` finds all of the unique values of `<state>` for us, we don't have to worry about finding them ourselves. If we did need to find the set of unique values, XPath 2.0 and XQuery 1.0 provide the `distinct-values( )` function. Here's how we would get a list of unique states with XSLT 2.0:

```
<xsl:variable name="list-of-unique-states" as="xs:string*">
  <xsl:perform-sort>
    <xsl:sort select="states:getStateName(.)"/>
    <xsl:sequence
      select="distinct-values(document(/report/po/@filename)
              /purchase-order/customer/address/state)"/>
```

```
    </xsl:perform-sort>
  </xsl:variable>
```

We create a variable of type `xs:string*`. That means our variable will be a sequence of strings, each of which is a unique state abbreviation. Next we use the new `<xsl:perform-sort>` element to sort our unique values by state name, not abbreviation. Inside `<xsl:perform-sort>`, we have the `<xsl:sort>` element, which sorts all of the values by state name, and the `<xsl:sequence>` element, which selects the distinct values of all the `<state>` elements. (We're using the `states:getStateName( )` function.)

It's great that XSLT 2.0 gives us this powerful yet simple technique; it's even better that `<xsl:for-each-group>` simplifies things so we don't have to use it here at all.

Doing Math Without Recursion

Our original stylesheet had to use recursion to calculate the total of each purchase order. We passed all of the `<item>` elements to a recursive template, which returned the total for the purchase order after processing all of the `<item>`s. In XSLT 2.0, we can use a simple XPath expression to do all that work for us. Here's the single line of code:

```
<xsl:value-of
  select="format-number(sum(items/item/(qty * price)),
          '$#,###.00')"/>
```

That's it. XPath 2.0's support for mathematical expressions inside path expressions makes this ridiculously easy. We replaced maybe 20 lines of code with just 1.

Implementing Lookup Tables with <xsl:function>

Using `document('')` to implement lookup tables is a workable approach, but we can simplify things with an XSLT function. With a function, we can invoke `getStateName( )` to resolve a state abbreviation to a state name; that's much simpler than invoking `document('')/*/states:name[@abbrev=current( )]` or something similar. Here's how our function looks:

```
<xsl:function name="states:getStateName" as="xs:string">
  <xsl:param name="abbr" as="xs:string"/>
  <xsl:variable name="abbreviations" as="xs:string*"
    select="'AL', 'AK', 'AS', 'AZ', 'AR', 'CA', 'CO', 'CT',
<!-- state abbreviations removed for brevity -->
            'WV', 'WI', 'WY'"/>
  <xsl:variable name="stateNames" as="xs:string*"
    select="'Alabama', 'Alaska', 'American Samoa','Arizona',
<!-- state names removed for brevity -->
            'West Virginia', 'Wisconsin', 'Wyoming'"/>
  <xsl:variable name="index"
    select="if (count(index-of($abbreviations, $abbr)) gt 0)
            then subsequence(index-of($abbreviations, $abbr), 1, 1)
            else 0"/>
```

```
    <xsl:value-of
      select="if ($index gt 0)
              then string(subsequence($stateNames, $index, 1))
              else ''"/>
  </xsl:function>
```

Our function takes a single string, a state abbreviation, as a parameter, and it returns a single string. It uses two variables to store our data; `$abbreviations` is a sequence of all state abbreviations, and `$stateNames` is a sequence of all state names. The positions of items in the two sequences match each other. If the abbreviation passed to the function matches the fifth item in the `$abbreviations` sequence, the name of the state is the fifth item in the `$stateNames` sequence.

To calculate the position of the matching abbreviation, we use the XPath 2.0 and XQuery 1.0 `index-of( )` function. This returns a sequence of `xs:integer`s, each of which represents the index of a match. We use the `count( )` function to count the number of items in the sequence returned by `index-of( )`. If the abbreviation passed to the function matches something in the `$abbreviations` sequence, the size of the sequence will be greater than zero (actually, we know the size will be either one or zero). To get the index of the matching item, we have to use the new `subsequence( )` function to retrieve the first item in the sequence. If none of the abbreviations match the parameter, the index is zero.

Having calculated the index of the matching abbreviation, we return the appropriate state name. If the value of `$index` is greater than zero, we use `subsequence( )` to return the appropriate item from the `$stateNames` sequence. If the `$index` is zero (we get a state abbreviation we've never heard of), we return an empty string.

Notice that we simplify our code here by using the XPath 2.0 and XQuery 1.0 `if` operator. We could do this with `<xsl:choose>`, `<xsl:when>`, and `<xsl:otherwise>`, but `if` is much simpler. Also notice that we used the `gt` comparison operator; we can do that because we're comparing two atomic values here. We could have used `>` or `>` with the same results.

Now that we have our function defined, `states:getStateName('AL')` returns `Alabama`. We can use this greatly simplified syntax wherever we need it. This puts all of the logic for resolving state abbreviations into one place in the stylesheet, instead of forcing us to use `document('')` with the appropriate XPath expression wherever we need it.

Using if Instead of <xsl:choose>

Although the XSLT elements `<xsl:if>`, `<xsl:choose>`, `<xsl:when>`, and `<xsl:otherwise>` provide the same function as the if-then-else functions of most programming languages, they're much more verbose. We can use the XPath 2.0 and XQuery 1.0 `if` operator to simplify things.

Our XSLT 1.0 stylesheet uses `<xsl:choose>` in two places: to choose the background color of table rows, and to change the decimal format for currency amounts in the first row of each purchase order. Here's the first `<xsl:choose>`:

```
<tr>
  <xsl:attribute name="style">
    <xsl:text>background: </xsl:text>
    <xsl:choose>
      <xsl:when test="position() mod 2">
        <xsl:text>#CCCCFF</xsl:text>
      </xsl:when>
      <xsl:otherwise>
        <xsl:text>#66FF66</xsl:text>
      </xsl:otherwise>
    </xsl:choose>
    <xsl:text>;</xsl:text>
  </xsl:attribute>
```

We create an attribute named `style`. If this is an odd-numbered row (`position( ) mod 2` is `1`, which evaluates to `true`), we set the background color to `#CCCCFF`; otherwise, we set it to `#66FF66`. In our XSLT 2.0 stylesheet, we can replace those elements with this attribute value template:

```
<tr style="{if (position() mod 2)
            then 'background: #CCCCFF;'
            else 'background: #66FF66;'}">
```

The other place we use `<xsl:choose>` is to choose a decimal format. Here's how we do it in XSLT 2.0:

```
<xsl:value-of
  select="if (position() = 1)
          then format-number(price * qty, '$#,###.00')
          else format-number(price * qty, '#,###.00')"/>
```

In both cases, we've replaced several elements with a single `if` statement inside an attribute.

Using the format-date( ) Function

Our last enhancement is to use the new `format-date( )` function to format the date of the purchase order. This is an addition to the function of our XSLT 1.0 stylesheet; everything else we've done here has duplicated XSLT 1.0 function in a much simpler way. The date of each purchase order is stored as three attributes named `year`, `month`, and `day`. We'll use the values of those three attributes to create a new `xs:date` value, then we'll use `format-date( )` to format the value.

To create a new `xs:date` value, we need a string in the format `yyyy-mm-dd`. `xs:date('2006-10-10')` is a valid call to the `xs:date` constructor. To complicate things, the month and day values *must* have two digits. In other words, `xs:date('2006-9-8')` raises an error. Our purchase orders don't necessarily have two-digit month and day

values, so we'll have to write code to add a leading zero if either value is less than 10. Here's how we do this:

```
<xsl:variable name="monthValue" as="xs:string"
  select="if (date/@month &lt; 10)
          then concat('0', date/@month)
          else date/@month"/>
<xsl:variable name="dayValue" as="xs:string"
  select="if (date/@day &lt; 10)
          then concat('0', date/@day)
          else date/@day"/>
```

We compare the value of the attributes to `10`; if they're smaller, we add the string `'0'` to the value. (Notice that the two variables we're creating here are of datatype `xs:string`.) In this code, we have to use the `<` operator because we're comparing a node to a number. If we want to use the `lt` operator, we'd have to code `if (number(date/@day) lt 10)`.

Once we've normalized the month and day values, we can create our `xs:date` value and format it. Here's the rest of the code:

```
<xsl:variable name="orderDate" as="xs:date"
  select="xs:date(concat(date/@year, '-',
          $monthValue, '-', $dayValue))"/>
<xsl:value-of
  select="format-date($orderDate, '[FNn], [MNn] [D1], [Y]')"/>
```

The date-formatting codes here format the date `2006-10-10` as `Tuesday, October 10, 2006`. To do this in XSLT 1.0, we would have to create a lookup table to convert month numbers to their text equivalents. We would also need a lookup table for the names of the days of the week, although we'd have to write an extension function to figure out that the 10th of October, 2006 occurred on a Tuesday.

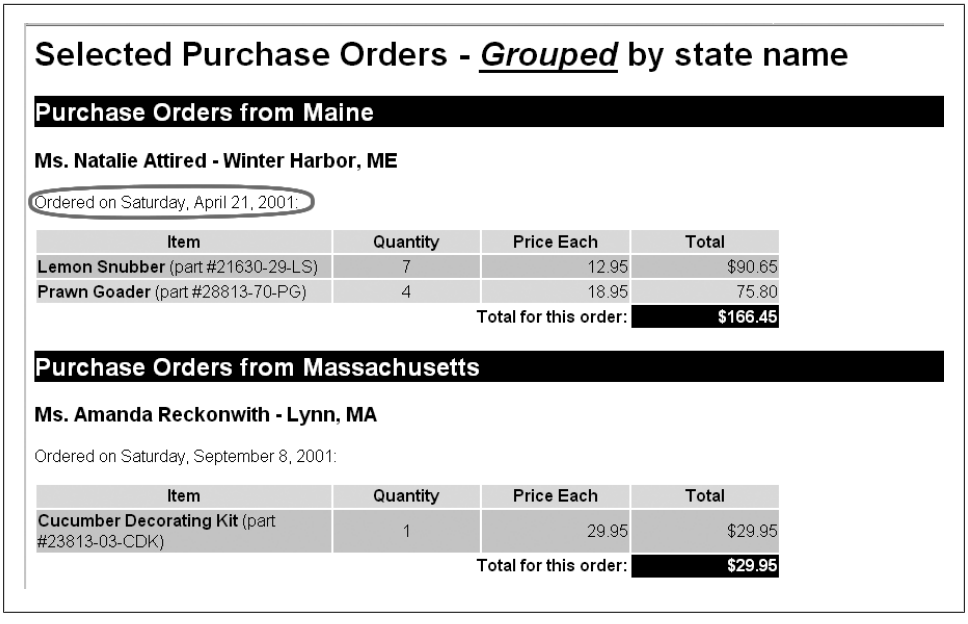
The Complete XSLT 2.0 Solution

Here's how our final stylesheet looks in XSLT 2.0:

```
<?xml version="1.0"?>
<!-- masterdox5.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:states="http://www.usps.com/ncsc/lookups/abbreviations.html"
  exclude-result-prefixes="xs states">

  <xsl:variable name="purchase-orders"
    select="document(/report/po/@filename)/purchase-order"/>

  <xsl:function name="states:getStateName" as="xs:string">
    <xsl:param name="abbr" as="xs:string"/>
    <xsl:variable name="abbreviations" as="xs:string*"
      select="'AL', 'AK', 'AS', 'AZ', 'AR', 'CA', 'CO', 'CT',
    <!-- state abbreviations removed for brevity -->
```

```
            'WV', 'WI', 'WY'"/>
  <xsl:variable name="stateNames" as="xs:string*"
    select="'Alabama', 'Alaska', 'American Samoa','Arizona',
<!-- state names removed for brevity -->
            'West Virginia', 'Wisconsin', 'Wyoming'"/>
  <xsl:variable name="index"
    select="if (count(index-of($abbreviations, $abbr)) gt 0)
            then subsequence(index-of($abbreviations, $abbr), 1, 1)
            else 0"/>
  <xsl:value-of
    select="if ($index gt 0)
            then string(subsequence($stateNames, $index, 1))
            else ''"/>
</xsl:function>

<xsl:output method="html"/>

<xsl:template match="/">
  <html>
    <head>
      <title><xsl:value-of select="/report/title"/></title>
    </head>
    <body style="font-family: sans-serif;">
      <h1>
        Selected Purchase Orders -
        <span style="font-style: italic;
                     text-decoration: underline;">Grouped</span>
        by state name
      </h1>
      <xsl:for-each-group
        select="$purchase-orders" group-by="customer/address/state">
        <xsl:sort
          select="states:getStateName(current-grouping-key())"/>
        <h2 style="color: white; background: black;">
          Purchase Orders from
          <xsl:value-of
            select="states:getStateName(current-grouping-key())"/>
        </h2>
        <xsl:for-each select="current-group()">
          <xsl:sort select="customer/address/city"/>
          <xsl:apply-templates select="."/>
        </xsl:for-each>
      </xsl:for-each-group>
    </body>
  </html>
</xsl:template>

<xsl:template match="purchase-order">
  <h3>
    <xsl:value-of
      select="customer/address[@type='business']/name/title"/>
    <xsl:text> </xsl:text>
    <xsl:value-of
      select="customer/address[@type='business']/name/first-name"/>
    <xsl:text> </xsl:text>
```

```
  <xsl:value-of
    select="customer/address[@type='business']/name/last-name"/>
  <xsl:text> - </xsl:text>
  <xsl:value-of
    select="customer/address[@type='business']/city"/>
  <xsl:text>, </xsl:text>
  <span style="font-weight: bold;">
    <xsl:value-of
      select="customer/address[@type='business']/state"/>
  </span>
</h3>
<p>
  <xsl:text>Ordered on </xsl:text>
  <xsl:variable name="monthValue" as="xs:string"
    select="if (date/@month &lt; 10)
            then concat('0', date/@month)
            else date/@month"/>
  <xsl:variable name="dayValue" as="xs:string"
    select="if (date/@day &lt; 10)
            then concat('0', date/@day)
            else date/@day"/>
  <xsl:variable name="orderDate" as="xs:date"
    select="xs:date(concat(date/@year, '-',
                           $monthValue, '-', $dayValue))"/>
  <xsl:value-of
    select="format-date($orderDate, '[FNn], [MNn] [D1], [Y]')"/>
  <xsl:text>:</xsl:text>
</p>
<table width="80%" border="0">
  <tr style="background: #66FF66;">
    <th>Item</th>
    <th>Quantity</th>
    <th>Price Each</th>
    <th>Total</th>
  </tr>
  <xsl:for-each select="items/item">
    <tr style="{if ((position() mod 2) = 1)
                then 'background: #CCCCFF;'
                else 'background: #66FF66;'}">
      <td width="40%">
        <span style="font-weight: bold;">
          <xsl:value-of select="name"/>
        </span>
        <xsl:text> (part #</xsl:text>
        <xsl:value-of select="@part-no"/>
        <xsl:text>)</xsl:text>
      </td>
      <td style="text-align: center;" width="20%">
        <xsl:value-of select="qty"/>
      </td>
      <td style="text-align: right;" width="20%">
        <xsl:value-of select="price"/>
      </td>
      <td style="text-align: right;" width="20%">
        <xsl:value-of
```



Selected Purchase Orders - *Grouped* by state name

Purchase Orders from Maine

Ms. Natalie Attired - Winter Harbor, ME

Ordered on Saturday, April 21, 2001:

Item	Quantity	Price Each	Total
Lemon Snubber (part #21630-29-LS)	7	12.95	$90.65
Prawn Goader (part #28813-70-PG)	4	18.95	75.80
		Total for this order:	**$166.45**

Purchase Orders from Massachusetts

Ms. Amanda Reckonwith - Lynn, MA

Ordered on Saturday, September 8, 2001:

Item	Quantity	Price Each	Total
Cucumber Decorating Kit (part #23813-03-CDK)	1	29.95	$29.95
		Total for this order:	**$29.95**

Figure 8-5. Purchase orders grouped by state name—XSLT 2.0 version

```
                select="if (position() = 1)
                        then format-number(price * qty, '$#,###.00')
                        else format-number(price * qty, '#,###.00')"/>
            </td>
          </tr>
        </xsl:for-each>
        <tr style="font-weight: bold;">
          <td colspan="3" style="text-align: right;">
            Total for this order:
          </td>
          <td align="right" style="color: white; background: black;">
            <xsl:value-of
              select="format-number(sum(items/item/(qty * price)),
                                    '$#,###.00')"/>
          </td>
        </tr>
      </table>
    </xsl:template>

</xsl:stylesheet>
```

The results of our stylesheet are as shown in Figure 8-5.

Notice that the dates are formatted differently than they were in the XSLT 1.0 version; with that exception, the output is identical, even though the stylesheet is much shorter.

[2.0] The doc( ) and doc-available( ) Functions

XSLT 2.0 provides a new function, `doc( )`, that is very similar to the `document( )` function, but that is simpler in a couple of ways. First of all, the `document( )` function can take a node-set or sequence as its first argument, whereas `doc( )` takes a single string. In our earlier stylesheets, we created a variable containing all of the nodes from all of the documents referenced in our list of purchase orders:

```
<xsl:variable name="purchase-orders"
  select="document(/report/po/@filename)/purchase-order"/>
```

This returns a sequence of `purchase-order` nodes, each of which has the structure defined in the purchase order. We can use the variable `$purchase-orders` in an `<xsl:for-each>` or `<xsl:for-each-group>` element. A stylesheet that contains this markup raises an error if there is more than one `<po>` element in the source document:

```
<!-- This doesn't work with our sample document -->
<xsl:variable name="purchase-orders"
  select="doc(/report/po/@filename)/purchase-order"/>
```

If we want to use the `doc( )` function, there are several things we could do. We could specify a particular `<po>` element:

```
<xsl:variable name="purchase-orders"
  select="doc(/report/po[2]/@filename)/purchase-order"/>
```

A more practical solution would be to put the `doc( )` function into an `<xsl:for-each>` element:

```
<xsl:for-each select="/report/po">
  <xsl:variable name="purchase-orders"
    select="doc(@filename)/purchase-order"/>
</xsl:for-each>
```

We could also change our input document so that it has only a single `<po>` element; although that's hardly a worthwhile solution, the `doc( )` function would in fact work.

To contrast the two functions, the `doc( )` function returns the document node of a single XML document, while the `document( )` function can return a set of nodes from multiple documents. In most cases, you'll want to use the more flexible `document( )` function instead.

Second, we mentioned that the `document( )` function has an optional second argument to set a base URI for resolving relative URLs. *The* `doc( )` *function doesn't have this option*; the base URI is always the base URI of the stylesheet.

The final difference between `doc( )` and `document( )` is that the `document( )` function supports fragment identifiers in the URL, whereas *the* `doc( )` *function does not*. If your processor uses fragment identifiers, the `document( )` function can return specific nodes in a document rather than simply returning the document node.

```
<!-- This is legal: -->
<xsl:variable name="mainContent"
```

```
  select="document('http://www.ibm.com/developerworks/index.html#main')"/>

<!-- This is not: -->
<xsl:variable name="mainContent"
  select="doc('http://www.ibm.com/developerworks/index.html#main')"/>
```

Here's a small excerpt of our first stylesheet (`masterdox1.xsl`), rewritten to use `doc( )` instead of `document( )`:

```
<?xml version="1.0"?>
<!-- doc.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title><xsl:value-of select="/report/title"/></title>
      </head>
      <body style="font-family: sans-serif;">
        <h1>Selected Purchase Orders - Unsorted</h1>
        <xsl:for-each select="/report/po">
          <xsl:apply-templates
            select="doc(@filename)/purchase-order"/>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>
...
```

We've structured our stylesheet so that each `filename` attribute is passed to the `doc( )` function as a single string. This gives us the same results as the original stylesheet; rewriting this chapter's more complicated stylesheets to use `doc( )` instead of `document( )` would be more difficult.

As a companion to the `doc( )` function, XPath 2.0 and XQuery 1.0 provide the `doc-available( )` function. This returns `true` if a given URL is available; it returns `false` otherwise. Here's a simple stylesheet that uses this function:

```
<?xml version="1.0"?>
<!-- doc-available.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the doc-available() function:&#xA;</xsl:text>

    <xsl:text>&#xA;  doc-available('polist.xml') = </xsl:text>
    <xsl:value-of select="doc-available('polist.xml')"/>

    <xsl:text>&#xA;&#xA;  doc-available('polist2.xml') = </xsl:text>
```

```
    <xsl:value-of select="doc-available('polist2.xml')"/>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates these results:

```
Tests of the doc-available() function:

  doc-available('polist.xml') = true

  doc-available('polist2.xml') = false
```

You can use the `doc-available( )` function to see whether opening a particular URL would raise an error. If `doc-available( )` returns `true`, it's safe to use the `doc( )` function to open the requested URL.

Be aware that the argument to `doc-available( )` has the same restrictions as the argument to `doc( )`. If you pass an argument to `doc-available( )` that isn't legal for `doc( )` (the aforementioned expression `doc(/report/po/@filename)/purchase-order`, for example), `doc-available( )` returns `false`.

To sum it up, if `doc-available( )` returns `true`, then `document( )` will work; on the other hand, if `doc-available( )` returns `false`, it's possible that `document( )` will still work. XSLT 2.0 doesn't have a `document-available( )` function.

[2.0] The collection( ) Function

The `collection( )` function takes a string as its argument and returns a collection of nodes. Defined as part of the XPath 2.0 spec, it gives us the ability to use a URI to retrieve a collection of documents. How those documents are stored (or whether they're really documents at all) is implementation-dependent. In particular, the spec mentions accessing data in a relational database as a possible implementation of the `collection( )` function. The fact that the string passed to the function can be generated and can contain parameters makes `collection( )` very flexible.

We'll look at a short example here. Here's the document we'll pass to the `collection( )` function:

```
<?xml version="1.0"?>
<!-- polist.xml -->
<collection>
  <doc href="po38292.xml"/>
  <doc href="po38293.xml"/>
  <doc href="po38294.xml"/>
  <doc href="po38295.xml"/>
</collection>
```

This is very similar to the list of purchase orders we worked with earlier in this chapter. The stylesheet that invokes the `collection( )` function looks like this:

```
<?xml version="1.0"?>
<!-- collection.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the collection() function:</xsl:text>

    <xsl:variable name="docPile" as="node()*"
      select="collection('polist.xml')"/>

    <xsl:text>&#xA;&#xA;  The customers in the </xsl:text>
    <xsl:text>collection are: &#xA;    </xsl:text>
    <xsl:for-each select="$docPile/purchase-order/customer">
      <xsl:sort select="address/name/last-name"/>
      <xsl:value-of
        select="address/name/title,
                address/name/first-name,
                address/name/last-name"
        separator=" "/>
      <xsl:text>    &#xA;    </xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet extracts the nodes from the `collection( )` function and stores them in the variable `$docPile`. Once we have the nodes from the collection, we sort all of the customers in all of those purchase orders by last name, and then write them out. The results look like this:

```
A test of the collection() function:

  The customers in the collection are:
    Ms. Natalie Attired
    Mrs. Mary Backstayge
    Mr. Chester Hasbrouck Frisby
    Ms. Amanda Reckonwith
```

See the definition of the [2.0] `collection( )` function in Appendix C for a more complete discussion.

[2.0] The unparsed-text( ) and unparsed-text-available( ) Functions

The last new function for combining documents is the `unparsed-text( )` function. This lets you read in text from a URL. That text is not parsed, letting you read in text

documents, comma-separated values, or even HTML documents that aren't well-formed XML. What's more, you can combine `unparsed-text( )` with other new features such as the `tokenize( )` function or the `<xsl:analyze-string>` element to process that text and transform it in a useful way.

As an example, we'll read in a file of comma-separated values and output them as an HTML table of addresses. Here's the comma-separated file, *unparsed-text.csv*:

```
Mr.,Chester Hasbrouck,Frisby,1234 Main Street,Sheboygan,WI,48392
Ms.,Natalie,Attired,707 Breitling Way,Winter Harbor,ME,00218
Ms.,Amanda,Reckonwith,930-A Chestnut Street,Lynn,MA,02930
Mrs.,Mary,Backstayge,283 First Avenue,Skunk Haven,MA,02718
```

We'll go through three simple steps to process this data. First, we'll use the `tokenize( )` function to get each line of the file. Next, we'll use `tokenize( )` to get each comma-separated value. Finally, we'll take each value and transform it appropriately. Using the comma-separated file we've listed here, the third comma-separated value in each line is the customer's last name, the seventh value is the zip code, and so forth.

To process the file one line at a time, we'll use this technique, courtesy of the XSLT 2.0 spec:

```
<xsl:for-each
  select="tokenize(unparsed-text('addresses.csv'), '\r?\n')">
```

The `<xsl:for-each>` element processes the file one line at a time, while the regular expression `\r?\n` matches a line end. (As the spec points out, `unparsed-text( )` doesn't normalize line endings, so we have to allow for an optional carriage return, indicated by `\r?`.)

Within `<xsl:for-each>`, we tokenize each line. This is easy because we're simply looking for the values between the commas. The `tokenize( )` function returns a sequence, so we put that sequence into a variable:

```
<xsl:variable name="tokens" select="tokenize(., ',')"/>
```

Before we process the comma-separated values, we use the `count( )` function to make sure the tokenizer found anything. If the CSV file contains a blank line, the number of tokens for that line will be zero; we'll want to ignore that line and move on to the next one. The rest of the stylesheet uses the `subsequence( )` function to retrieve the particular values we want.

Here's the complete stylesheet that processes the CSV file:

```
<?xml version="1.0"?>
<!-- unparsed-text.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
```

```
    <head>
      <title>Customer Addresses</title>
    </head>
    <body style="font-family: sans-serif;">
      <h1>Customer Addresses</h1>
      <table width="60%" border="0">
        <tr style="background: #66FF66;">
          <th>Name</th>
          <th>Street</th>
          <th>City</th>
          <th>State</th>
          <th>Zip</th>
        </tr>
        <xsl:for-each
          select="tokenize(unparsed-text('addresses.csv'), '\r?\n')">
          <xsl:variable name="tokens" select="tokenize(., ',')"/>
          <xsl:if test="count($tokens)">
            <tr style="{if (position() mod 2)
                        then 'background: #CCCCFF;'
                        else 'background: #66FF66;'}">
              <td width="40%">
                <xsl:value-of
                  select="subsequence($tokens, 1, 2)" separator=" "/>
                <xsl:text> </xsl:text>
                <span style="font-weight: bold; font-size: 125%;">
                  <xsl:value-of select="subsequence($tokens, 3, 1)"/>
                </span>
              </td>
              <td width="20%">
                <xsl:value-of select="subsequence($tokens, 4, 1)"/>
              </td>
              <td width="20%">
                <xsl:value-of select="subsequence($tokens, 5, 1)"/>
              </td>
              <td width="10%" style="text-align: center;">
                <xsl:value-of select="subsequence($tokens, 6, 1)"/>
              </td>
              <td width="10%" style="text-align: center;">
                <xsl:value-of select="subsequence($tokens, 7, 1)"/>
              </td>
            </tr>
          </xsl:if>
        </xsl:for-each>
      </table>
    </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

The results look like Figure 8-6.

Notice that we selected the first two tokens (the customer's courtesy title and first name) as a sequence, and then wrote them out with the attribute `separator=" "`. If a customer

Customer Addresses

Name	Street	City	State	Zip
Mr. Chester Hasbrouck **Frisby**	1234 Main Street	Sheboygan	WI	48392
Ms. Natalie **Attired**	707 Breitling Way	Winter Harbor	ME	00218
Ms. Amanda **Reckonwith**	930-A Chestnut Street	Lynn	MA	02930
Mrs. Mary **Backstayge**	283 First Avenue	Skunk Haven	MA	02718

Figure 8-6. HTML document generated from a file of comma-separated values

doesn't have a courtesy title, our stylesheet doesn't insert an unnecessary space before the customer's first name.

The `unparsed-text( )` function has an optional second argument to specify the encoding of the document located at the URL. For example, if the file *addresses.csv* used the UTF-16 encoding, we would use `unparsed-text('addresses.csv', 'utf-16')` to read the file.

You can use the `unparsed-text( )` function to import an XML or HTML file without parsing it. See the definition of the `unparsed-text( )` function for more examples.

Analogous to the `doc-available( )` function is the new `unparsed-text-available( )` function. It returns `true` if the unparsed text located at a given URL is available; it returns `false` otherwise.

Summary

This chapter completes our tour of the XSLT and XPath functions that work with multiple documents. These powerful functions allow us to generate an output document containing elements from many different input documents. In our examples here, we generated several views of those input documents, but many more combinations might be useful. The biggest benefit of these functions is that they allows us to define views of multiple documents that are separate from those documents themselves. As we need to define other views, we don't have to change our input documents.

We also looked at using the new `unparsed-text( )` function to read non-XML data and add it to our result documents. The functions we've discussed here can save you a tremendous amount of development time in generating reports and other summarizing documents.

CHAPTER 9

Extending XSLT

To this point, we've spent a lot of time learning how to use the built-in features of XSLT and XPath to get things done. We've also talked about the somewhat unusual processing model that makes life challenging for programmers from the world of procedural languages (a.k.a. Earth). But what do you do if you still can't do everything with XSLT and XPath?

In this chapter, we'll discuss the XSLT extension mechanism that allows you to add new functions and elements to the language. The XSLT standard doesn't define all of the details about how these things should work, so there are some differences between processors. The good news is that if you write an extension function or element that works with your favorite processor, another vendor can't do something sinister to prevent your functions or elements from working. The less good news is that if you decide to change XSLT processors, you'll probably have to change your code.

Along the way, we'll also discuss the EXSLT project, whose goals are to provide a common library of extension functions that work across different XSLT processors.

The examples in this chapter are written for the Java-based Saxon and Xalan processors, and for the .NET framework (using C#). We'll discuss how to write stylesheets that can work with multiple processors, and we'll briefly look at the differences between the various APIs supported by those processors. In addition, the Xalan-J processor supports Apache's Bean Scripting Framework (BSF), which means we can write extensions in Jython (also known as JPython), JavaScript, Jacl, and any other language supported by the BSF.

The XSLT Extension Mechanism

The XSLT standard defines two kinds of extensions: extension elements and extension functions. The spec also defines *fallback processing*, a way for stylesheets to respond gracefully when extension elements and functions aren't available. (Fallback processing also applies when we ask an XSLT 1.0 processor to process an XSLT 2.0 stylesheet.) We'll talk about these items briefly, and then we'll move on to some examples that illustrate the full range of extensions and fallback processing.

Keep in mind that the XSLT specs define how the extension mechanism should work; they *do not* define how it should be implemented. As we'll see throughout this chapter, different XSLT processors implement extensions in different ways. For example, a Java-based XSLT processor might use a Java class to implement an extension function or extension element, while a .NET XSLT processor might use a .NET class to do the same thing. How those classes are specified varies from one processor to the next.

Extension Elements

An extension element is an element that should be processed by a piece of code external to the XSLT processor. The implementation details vary from one XSLT processor to the next, as you'd expect. We'll discuss how an extension element can access the XPath representation of our XML source document, how (or if) it can process attribute value templates, how it can generate output, and how (or if) it can move through the XPath tree to manipulate the source document. We'll demonstrate various APIs to do all of these things; the APIs, of course, vary quite a bit between processors. Finally, although XSLT processors typically provide an extension writer with access to the XML source, the standard doesn't define a set of functions or access methods that must be supported.

Extension Functions

As you might guess, an extension function is defined in a piece of code accessed from the XSLT stylesheet. In some cases, that function is written in a scripting language in the stylesheet itself. In those instances, the scripting code is contained in an XML element that is not in the XSLT namespace. In other cases, the function is in a separate file that is loaded by the XSLT processor at runtime. The code for the function might be written in a scripting language or it might be written in a compiled language.

Regardless of how the extension function is written or accessed, you can pass values to the function and the function can return a result. That result can be any of the datatypes supported by XPath. In addition, XSLT processors are free to allow extension functions to return other datatypes, although those other datatypes must be handled by some other function that does return one of XPath's datatypes.

Fallback Processing

The final part of XSLT's extension mechanism uses the `<xsl:fallback>` element. This element is processed whenever an XSLT processor doesn't support the surrounding element. For example, the `<xsl:result-document>` element is new in XSLT 2.0. We can put an `<xsl:fallback>` element inside `<xsl:result-document>`. If we pass the stylesheet to an XSLT 1.0 processor, it won't support the `<xsl:result-document>` element. The XSLT 1.0 processor will look inside the `<xsl:result-document>` for an `<xsl:fallback>` element; if it finds one, it will process the contents of `<xsl:fallback>`. As we'll see a little later in this chapter, `<xsl:fallback>` can itself contain an `<xsl:fallback>`

element. The <xsl:fallback> element gives the XSLT processor a chance to respond gracefully when it can't perform the requested task.

Namespaces for Extensions

XSLT extension elements and extension functions must have a namespace prefix different from the XSLT namespace prefix. For example, here's the start of a stylesheet that declares several namespace prefixes:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:svg="http://www.w3.org/2000/svg"
  xmlns:xalan-java="http://xml.apache.org/xslt/java"
  xmlns:saxon-java="java:java.lang.Math"
  extension-element-prefixes="xalan-java saxon-java">
```

By default, the XSLT processor copies all of the non-XSLT namespaces to the output document. That means the `svg`, `xalan-java`, and `saxon-java` namespaces will be defined in the output. In this example, we're generating an SVG document using extensions associated with the `xalan-java` and `saxon-java` namespaces. We don't need our extension namespaces defined in the output document, so we use the `extension-element-prefixes` attribute to make sure they aren't in the generated document. The `svg` namespace, on the other hand, will appear in the output document, which is exactly what we want.

[2.0] Creating New Functions with <xsl:function>

XSLT 2.0 adds the <xsl:function> element. This lets you define your own functions in the stylesheet itself. This is the simplest extension that we'll examine in this chapter; the extension itself is in the same file and same syntax as the stylesheet, and the standard is very clear on how the function is defined and invoked. We'll use a simple stylesheet that creates a table in which the background color of each cell cycles through four different colors. Given the position of the current item, our function will return one of the four values.

To define a function, there are several things we have to do: define a (non-XSLT) namespace for the function, name the function, define what datatype it returns, and define the name and datatype of any parameters the function has. Defining a namespace is simple enough, although we need to remember to put our namespace prefix in the `exclude-result-prefixes` attribute of <xsl:stylesheet>. We'll call our function `getBackgroundColor`, and it will return an `xs:string` naming the background color of each table cell. Finally, the input to our function is an `xs:integer` of the position of the current item. The function looks like this:

```
<xsl:function name="sample:getBackgroundColor" as="xs:string">
  <xsl:param name="pos" as="xs:integer"/>
```

```
  <xsl:value-of select="$colors[($pos mod count($colors)) + 1]"/>
</xsl:function>
```

The function is in the `sample` namespace, it returns a string, and it takes an integer as its only parameter. It references the variable `$colors` to retrieve a color name. Rather than use an XML document for input, we'll create a sequence of numbers and put each one in a table cell. Here's the stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- simple-function.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:sample="http://www.oreilly.com/catalog/xslt"
  exclude-result-prefixes="xs sample">

  <xsl:output method="html" include-content-type="no"/>

  <xsl:variable name="colors" as="xs:string *"
    select="('green', 'grey', 'blue', 'red')"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>A table with different background colors</title>
      </head>
      <body style="font-family: sans-serif;">
        <h1 style="font-size: 28;">
          A table with different background colors:
        </h1>
        <table border="3" cellpadding="5" cellspacing="5" width="50%">
          <tr>
            <xsl:for-each select="1 to 12">
              <td style="font-size: 48; color: black;
                         font-weight: bold; text-align: center;"
                bgcolor="{sample:getBackgroundColor(position())}">
                <xsl:value-of select="."/>
              </td>
            </xsl:for-each>
          </tr>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:function name="sample:getBackgroundColor" as="xs:string">
    <xsl:param name="pos" as="xs:integer"/>
    <xsl:value-of select="$colors[($pos mod count($colors)) + 1]"/>
  </xsl:function>

</xsl:stylesheet>
```

In this example, we use the XPath 2.0 `to` operator to create a sequence of values, and then we invoke the `getBackgroundColor` function to set the background color of each table cell. Figure 9-1 shows how the results look in a browser.

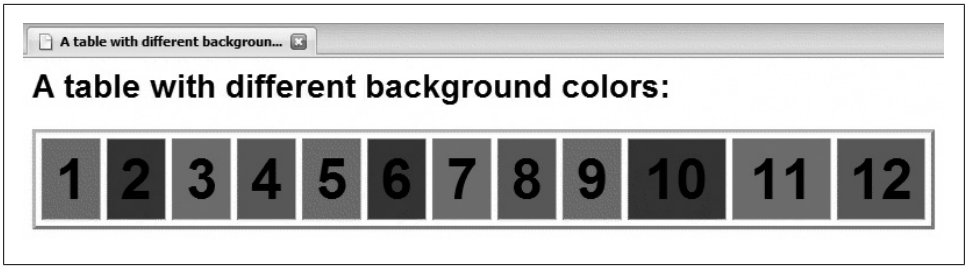


Figure 9-1. Results generated with <xsl:function>

To do the equivalent of this in XSLT 1.0, we would have to use logic like this:

```
<xsl:choose>
  <xsl:when test="position() mod 4 = 0">
    <!-- background color is xxx -->
  </xsl:when>
  <xsl:when test="position() mod 4 = 1">
    <!-- background color is xxx -->
  </xsl:when>
  ...
</xsl:choose>
```

This code is more verbose and is also less maintainable. If we change the number of colors in the sequence, we would have to change the XSLT 1.0 code. Our XSLT 2.0 function, on the other hand, uses the length of the sequence as part of its calculations, so we could add as many colors to the list as we'd like without changing the function code.

Example: Generating Multiple Output Files

The whole point of extensions is to allow you to add new capabilities to the XSLT processor. One of the most common needs is the ability to generate multiple output documents. As we saw earlier, the `document()` function allows you to have multiple input documents—but XSLT 1.0 doesn't give us any way to create these. Saxon's support for XSLT 2.0 includes the `<xsl:result-document>` element, which lets us generate multiple output documents. Although Xalan doesn't support XSLT 2.0, it does support an extension element (`<redirect:write>`) that does the same thing. We'll look at a stylesheet that uses `<xsl:fallback>` to generate useful results regardless of the processor we're using. If we're using Saxon or Xalan, we'll get multiple output documents that are hyperlinked together; if we're using any other processor, we'll get a single HTML file that contains the same information.

Here's the source document we'll use:

```
<?xml version="1.0"?>
<!-- chapters.xml -->
<book>
  <title>XSLT Topics</title>
  <chapter>
```

```
    <title>XPath</title>
    <para>If this chapter had any text, it would appear here.</para>
  </chapter>
  <chapter>
    <title>Stylesheet Basics</title>
    <para>If this chapter had any text, it would appear here.</para>
  </chapter>
...
  <chapter>
    <title>Combining XML Documents</title>
    <para>If this chapter had any text, it would appear here.</para>
  </chapter>
</book>
```

In addition to the `<xsl:fallback>` element, our stylesheet also uses the `element-available( )` function to determine what elements are available. If we can't generate multiple output documents (i.e., the elements we need aren't available), we create a single HTML file. If the elements are available, we first create a single HTML file with hyperlinks to the individual HTML files that we'll create later in the stylesheet. When we create the individual files, we'll use the `<xsl:result-document>` and `<redirect:write>` elements with `<xsl:fallback>` to handle with XSLT processors that don't support those elements.

Let's go through the relevant parts of this example. To begin with, our `<xsl:stylesheet>` element defines the `redirect` namespace prefix and tells the XSLT engine that the prefix will be used to refer to an extension element:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- multiple-output-files.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:redirect="org.apache.xalan.xslt.extensions.Redirect"
  extension-element-prefixes="redirect">
```

The `extension-element-prefixes` attribute tells the XSLT processor that any element or function name with the `redirect` namespace prefix should be handled by an extension. When the processor encounters something from this namespace, it loads a piece of code and tells it to handle the request from the stylesheet. If we don't tell the XSLT processor that `redirect` is an extension element prefix, Xalan-J throws an error.

The syntax of everything we've done so far is according to the standard, although there's a fair amount of latitude in what the XSLT engines do with the information we've defined. For example, when defining the `redirect` namespace, Xalan uses the value as a Java classname. In other words, Xalan attempts to load the class `org.apache.xalan.xslt.extensions.Redirect` when it encounters an extension element or function defined with this namespace.

In the first part of our stylesheet, we use the `element-available( )` function to see whether the XSLT processor supports the XSLT 2.0 element `<result-document>`. In the second part, we tell the processor what to do, and we include `<xsl:fallback>` elements in case the processor doesn't support XSLT 2.0.

The first part of the stylesheet looks like this:

```
<xsl:choose>
  <xsl:when test="element-available('xsl:result-document')">
    <-- Create an external link and a list item for each <chapter> -->
  </xsl:when>
  <xsl:when test="element-available('redirect:write')">
    <-- Create an external link and a list item for each <chapter> -->
  </xsl:when>
  <xsl:otherwise>
    <-- Create an internal link and a list item for each <chapter> -->
  </xsl:otherwise>
</xsl:choose>
```

For the first two branches, we generate a link of the form `<a href="chaptern.html">`, where *n* is the position of the current chapter. For the last branch, we generate a link of the form `<a href="#chaptern">`. If the processor can generate multiple output files, we'll generate a link to that separate file; otherwise, we'll create a link to a section of the current document.

The second part of the stylesheet looks like this:

```
<xsl:result-document method="html"
  include-content-type="no"
  href="{concat('chapter', position(), '.html')}">
  <-- Create a separate HTML file for each <chapter> -->
  <xsl:fallback>
    <redirect:write select="concat('chapter', position(), '.html')">
      <-- Create a separate HTML file for each <chapter> -->
      <xsl:fallback>
         <-- Create a named anchor, an <h1> and a paragraph >
      </xsl:fallback>
    </redirect:write>
  </xsl:fallback>
</xsl:result-document>
```

We ask the XSLT processor to use the `<xsl:result-document>` element. If we're not using an XSLT 2.0 processor, this won't work, so the processor tries to use the `<xsl:fallback>` element. If we're using Xalan-J, the `<redirect:write>` element is supported; if not, the processor uses the last `<xsl:fallback>` element.

If our XSLT processor can generate multiple output files, the HTML document appears as in Figure 9-2.

Clicking on any of the links here takes us to one of the generated files. Each separate file looks as shown in Figure 9-3.

If the XSLT processor can't generate multiple output files, we get a single file that contains all of the text of all of the chapters. The document starts with a set of links to each section in the document, as shown in Figure 9-4.

Here's the complete stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- multiple-output-files.xsl -->
```

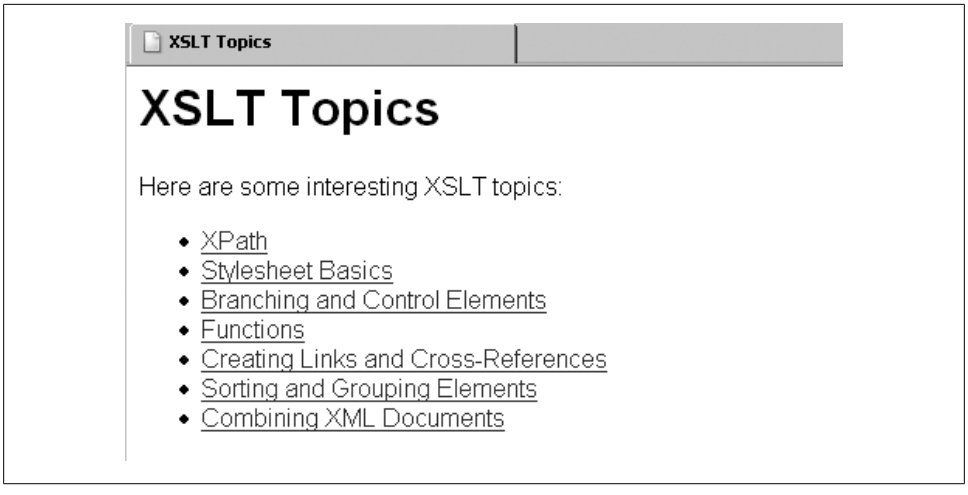


Figure 9-2. HTML file with links to multiple output files

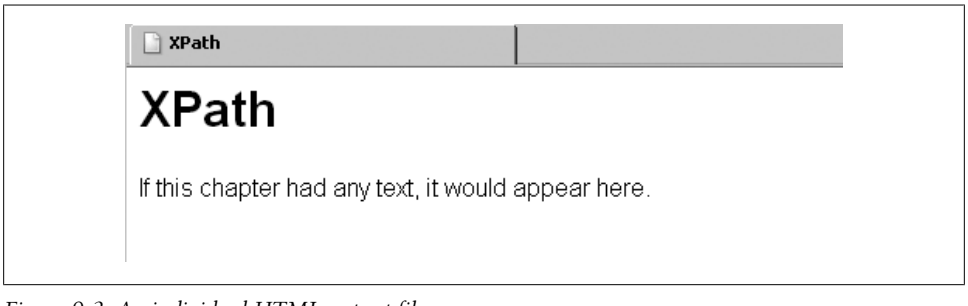


Figure 9-3. An individual HTML output file

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:redirect="org.apache.xalan.xslt.extensions.Redirect"
  extension-element-prefixes="redirect">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <xsl:comment>
          <xsl:text>Results generated by </xsl:text>
          <xsl:value-of select="system-property('xsl:vendor')"/>
        </xsl:comment>
        <title><xsl:value-of select="book/title"/></title>
      </head>
      <body style="font-family: sans-serif;">
        <h1><xsl:value-of select="book/title"/></h1>
        <p>Here are some interesting XSLT topics:</p>
        <ul>
```

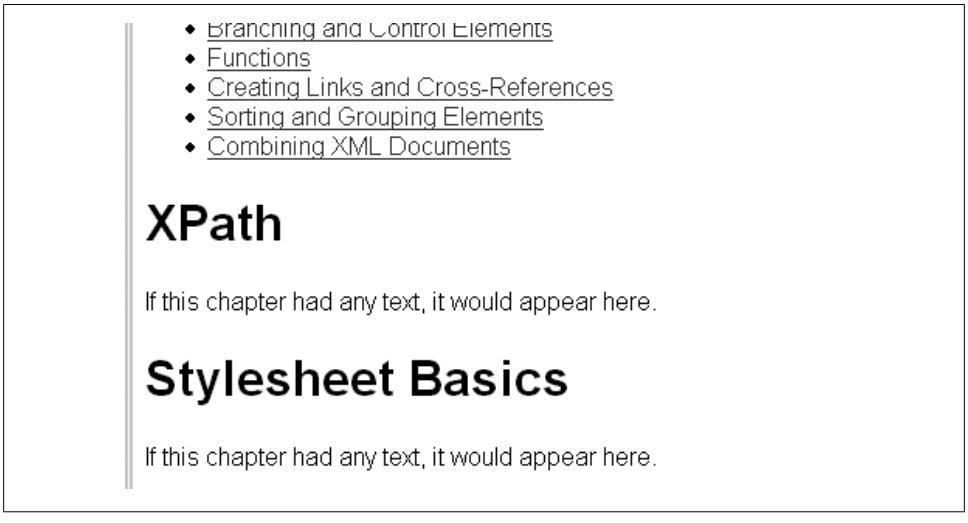

Branching and Control Elements
Functions
Creating Links and Cross-References
Sorting and Grouping Elements
Combining XML Documents
XPath
If this chapter had any text, it would appear here.
Stylesheet Basics
If this chapter had any text, it would appear here.

Figure 9-4. A single HTML output file that contains all the text of all the chapters

```
    <xsl:choose>
      <xsl:when test="element-available('xsl:result-document')">
        <xsl:apply-templates select="book/chapter" mode="separate-files"/>
      </xsl:when>
      <xsl:when test="element-available('redirect:write')">
        <xsl:apply-templates select="book/chapter" mode="separate-files"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:for-each select="book/chapter">
          <li>
            <a href="{concat('#chapter', position())}">
              <xsl:value-of select="title"/>
            </a>
          </li>
        </xsl:for-each>
      </xsl:otherwise>
    </xsl:choose>
  </ul>

  <xsl:for-each select="book/chapter">
    <xsl:result-document method="html"
      include-content-type="no"
      href="{concat('chapter', position(), ".html')}">
      <xsl:apply-templates select="." mode="create-html-file"/>

      <xsl:fallback>
        <redirect:write
          select="concat('chapter', position(), ".html')">
          <xsl:apply-templates select="." mode="create-html-file"/>

          <xsl:fallback>
            <a name="{concat('chapter', position())}"/>
```

```
                <h1>
                  <xsl:value-of select="title"/>
                </h1>
                <xsl:apply-templates select="*[position() > 1]"/>
              </xsl:fallback>
            </redirect:write>
          </xsl:fallback>
        </xsl:result-document>
      </xsl:for-each>
    </body>
  </html>
</xsl:template>

<xsl:template match="chapter" mode="separate-files">
  <li>
    <a href="{concat('chapter', position(), '.html')}">
      <xsl:value-of select="title"/>
    </a>
  </li>
</xsl:template>

<xsl:template match="chapter" mode="create-html-file">
  <html>
    <head>
      <title><xsl:value-of select="title"/></title>
    </head>
    <body style="font-family: sans-serif;">
      <h1><xsl:value-of select="title"/></h1>
      <xsl:apply-templates select="*[position() > 1]"/>
    </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

Notice that the stylesheet is a version 2.0 stylesheet. Telling an XSLT 1.0 processor that a stylesheet is an XSLT 2.0 stylesheet causes it to process the stylesheet in forwards-compatible mode. This means that it will ignore certain things that are errors in XSLT 1.0, and that it will go to the `<xsl:fallback>` element if it finds an element in the XSLT namespace that it doesn't recognize.

If you change the stylesheet to `version="1.0"`, Xalan (or MSXSL or whatever XSLT 1.0 processor you're using) raises a fatal error because the `<xsl:result-document>` element isn't defined in XSLT 1.0 either. If you're using *anything* from XSLT 2.0, be sure your `<xsl:stylesheet>` element clearly says so with the attribute `version="2.0"`.

In this relatively simple example, we've broken a single XML document into multiple HTML files, we've generated useful filenames for all of them, and we've automatically built hyperlinks to the different HTML files. For now, we've simply discussed how to *use* an extension; we'll talk about how to write your own extension next.

Creating Custom Collations

The XSLT 2.0 spec uses collations in several places. A collation defines how characters are sorted and compared. English doesn't have any accented characters or character sequences that sort as separate letters, so that's not an issue if all your documents are in English. Even if English is your native language, it's likely you'll need to work with documents written in other languages. In that case, characters such as the Spanish *ch* (considered a separate letter, the letter *che*) or accented characters such as the German umlaut-u, which can be written as *ü* or *ue*, become important in sorting and comparing words.

As with extension functions, the XSLT 2.0 spec defines attributes that can be used to indicate where custom collations can be used, but it doesn't define how to identify a particular piece of code that does the work. Because Saxon has taken the lead in implementing these functions, we'll focus on accessing custom collations in Saxon here. We'll look at two of these collations. The first sorts Spanish words so that *ch* sorts as a separate letter between *c* and *d*. The second collation compares German words so that *Müller* and *Mueller* are considered identical.

Your author is in no way a speaker of Spanish or German, so please pardon any incorrect statements about the languages themselves. The point here is to illustrate how to create extensions that implement custom collations and then use those extensions for sorting and comparing text in your stylesheets.

The traditional Spanish collation, the one we'll implement here, treats *ch*, *ll*, and *ñ* as separate letters that sort after *c*, *l*, and *n* respectively. However, much of the Spanish speaking world now uses the modern Spanish collation, defined by the Association of Spanish Language Academies (*La Asociación de Academias de Lingua Española*). The modern Spanish collation doesn't treat `ch` or `ll` as special characters; they sort as they would in English. The letter *ñ* still sorts after the letter *n*.

There are three collations for sorting German: DIN-1, DIN-2, and Austrian. (The DIN standards are defined by the standards body *Deutsches Institut Für Normung*.) The collation used varies from one country to the next. Typically DIN-1 is used for sorting words, although in Switzerland, it's also used for sorting names. DIN-2, the collation algorithm we'll implement here, is used for sorting names in Germany. Austria uses the Austrian collation, although it seems to be disappearing in favor of the DIN-2 rules. The main complication for the DIN-2 algorithm that we'll implement here is that `ä` is equal to `ae`, `ö` is equal to `oe`, `ß` is equal to `ss`, and `ü` is equal to `ue`. Our code has to realize that two characters in one word can be equivalent to one character in another word.

I believe the code samples here are correct implementations of the traditional Spanish and DIN-2 collation algorithms.

Using a Custom Collation for Sorting

To get started, we'll sort a short list of words:

```
<?xml version="1.0"?>
<!-- words.xml -->
<wordlist>
  <word>campo</word>
  <word>luna</word>
  <word>ciudad</word>
  <word>llaves</word>
  <word>chihuahua</word>
  <word>arroz</word>
  <word>limonada</word>
</wordlist>
```

This document contains Spanish words that are sorted differently than they would be in English. We'll write a stylesheet that uses two `<xsl:template>`s to illustrate how our extension function works. Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- custom-collation1.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Test of sorting with custom collations:&#xA;&#xA;</xsl:text>
    <xsl:variable name="items" as="xs:string*" select="wordlist/word"/>
    <xsl:text>Word list in original order:&#xA;&#xA;</xsl:text>
    <xsl:value-of select="$items" separator="&#xA;"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:call-template name="ascending-alpha-sort">
      <xsl:with-param name="items" select="$items"/>
    </xsl:call-template>
    <xsl:call-template name="spanish-alpha-sort">
      <xsl:with-param name="items" select="$items"/>
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="ascending-alpha-sort">
    <xsl:param name="items"/>
    <xsl:text>&#xA;Ascending text sort:&#xA;</xsl:text>
    <xsl:for-each select="$items">
      <xsl:sort/>
      <xsl:value-of select="."/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

  <xsl:template name="spanish-alpha-sort">
    <xsl:param name="items"/>
    <xsl:text>&#xA;Spanish alpha sort:&#xA;</xsl:text>
```

```
    <xsl:for-each select="$items">
      <xsl:sort
        collation="{concat('http://saxon.sf.net/collation?',
                   'class=com.oreilly.xslt.SpanishCollation;')}"/>
      <xsl:value-of select="."/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

We have to use an attribute value template as the value of the `collation` attribute because we used the `concat` function to keep the code listing within the page. If we had coded this in a single line (`collation="http://saxon.sf.net...."`), the curly braces would not be necessary. If you aren't worried about how your stylesheets look when printed out, simply put everything as a long string and avoid `concat` and the curly braces altogether.

We use Saxon's scheme for identifying the class that performs the custom collation. (We used `concat( )` to combine the two halves of the string so the listing fits on the page; normally you'd have the 70-character attribute value as a single string.) Saxon requires a particular URL, followed by a question mark and the keyword `class=`. Whatever follows `class=` is used as the name of the Java class. If that class can't be found or loaded, you'll get a fatal error. Here's the code for the extension that implements the custom collation:

```
/**
 * SpanishCollation.java
 */

package com.oreilly.xslt;

import java.text.ParseException;
import java.text.RuleBasedCollator;

public class SpanishCollation extends RuleBasedCollator
{
  public SpanishCollation() throws ParseException
  {
    super(traditionalSpanishRules);
  }

  private static String smallnTilde  = new String("\u00F1");
  private static String capitalNTilde = new String("\u00D1");

  private static String traditionalSpanishRules =
    ("< a,A < b,B < c,C < ch, cH, Ch, CH "  +
     "< d,D < e,E < f,F < g,G < h,H < i,I " +
     "< j,J < k,K < l,L < ll, lL, Ll, LL "  +
     "< m,M < n,N " +
```

```
        "< " + smallnTilde + "," + capitalNTilde + " " +
        "< o,O < p,P < q,Q < r,R < s,S < t,T " +
        "< u,U < v,V < w,W < x,X < y,Y < z,Z");
    }
```

The string `traditionalSpanishRules` contains the rules for character comparisons. Notice that it defines special cases for `ch`, `ll`, `Ñ`, and `ñ`. To create a custom collation, all we have to implement is the string that defines the collation rules, then pass that to the constructor of the `RuleBasedCollator` class. See the JDK documentation for all the details of the `java.text.RuleBasedCollator` class.

When we run the stylesheet against our document, it lists the words as they appear in the document, and then it calls two templates that sort the words. The first uses the default collation, so the words are sorted according to the rules of English. (English is the default collation on *my* machine; your machine might be different.) The second template invokes our Java class to do the comparisons between characters. The results are different for the two templates:

```
Test of sorting with custom collations:

Word list in original order:

campo
luna
ciudad
llaves
chihuahua
arroz
limonada

Ascending text sort:
arroz
campo
chihuahua
ciudad
limonada
llaves
luna

Spanish alpha sort:
arroz
campo
ciudad
chihuahua
limonada
luna
llaves
```

As you can see, sorting the word list with the `SpanishCollation` class gives us different results. `Chihuahua` sorts after `ciudad`, and `llaves` sorts after `luna`.

Using a Custom Collation for Comparing Text

The other way of using custom collations is in comparing text. Obviously comparing strings is part of sorting, but the new `compare( )` function in XQuery 1.0 and XPath 2.0 lets you compare two strings. To use an example from the XSLT 2.0 spec, the German word for *street* can be spelled *Straße* or *Strasse*. A simple character comparison defines sees these as two different strings; we'll use a collation function that recognizes the two words as being equal.

Here's the complete code:

```
/**
 * GermanCollation.java
 */

package com.oreilly.xslt;

import java.text.ParseException;
import java.text.RuleBasedCollator;

public class GermanCollation extends RuleBasedCollator
{
  public GermanCollation() throws ParseException
  {
    super(traditionalGermanRules);
  }

  private static String sharpS  = new String("\u00DF");
  private static String uppercaseUmlautA = new String("\u00C4");
  private static String lowercaseUmlautA = new String("\u00E4");
  private static String uppercaseUmlautO = new String("\u00D6");
  private static String lowercaseUmlautO = new String("\u00F6");
  private static String uppercaseUmlautU = new String("\u00DC");
  private static String lowercaseUmlautU = new String("\u00FC");

  private static String traditionalGermanRules =
    ("< a,A " +
     "<" + lowercaseUmlautA + "=ae " +
     "<" + uppercaseUmlautA + "=AE " +
     "< b,B < c,C < d,D < e,E < f,F " +
     "< g,G < h,H < i,I < j,J < k,K " +
     "< l,L < m,M < n,N < o,O " +
     "<" + lowercaseUmlautO + "=oe " +
     "<" + uppercaseUmlautO + "=OE " +
     "< p,P < q,Q < r,R < s,S " +
     "< ss=” + sharpS +
     "< t,T < u,U " +
     "<" + lowercaseUmlautU + "=ue " +
     "<" + uppercaseUmlautU + "=UE " +
     "< v,V < w,W < x,X < y,Y < z,Z");
}
```

In the code, we define that certain strings are equal. There are seven special rules *that are defined in* our class; each of them is defined with an equals sign in the

traditionalGermanRules string. (In the SpanishCollation class, we used a greater-than sign to indicate the sorting order.) Here is a stylesheet that compares two spellings of the German word for *street*:

```
<?xml version="1.0"?>
<!-- custom-collation2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="string1" select="'Stra&#xDF;e'"/>
    <xsl:variable name="string2" select="'Strasse'"/>

    <xsl:text>&#xA;Here is a test of the compare() </xsl:text>
    <xsl:text>function, using &#xA;</xsl:text>
    <xsl:text>  an extension function to compare German </xsl:text>
    <xsl:text>characters:&#xA;&#xA;</xsl:text>

    <xsl:text>  compare('</xsl:text>
    <xsl:value-of select="$string1"/>
    <xsl:text>', '</xsl:text>
    <xsl:value-of select="$string2"/>
    <xsl:text>') = </xsl:text>
    <xsl:value-of select="compare($string1, $string2)"/>
    <xsl:text>&#xA;</xsl:text>

    <xsl:text>  compare('</xsl:text>
    <xsl:value-of select="$string1"/>
    <xsl:text>', '</xsl:text>
    <xsl:value-of select="$string2"/>
    <xsl:text>', [German collation]) = </xsl:text>
    <xsl:value-of
      select="compare($string1, $string2,
              concat('http://saxon.sf.net/collation?',
                     'class=com.oreilly.xslt.GermanCollation;'))"/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Here is a test of the compare() function, using
  an extension function to compare German characters:

  compare('Straße', 'Strasse') = 1
  compare('Straße', 'Strasse', [German collation]) = 0
```

In the compare( ) function, a value of 1 means the first string sorts after the second, a value of -1 means the first string sorts before the second, and a value of 0 means the two strings are equal. In our results here, using the German collation indicates that

Straße and *Strasse* are identical, even though they're clearly two different sequences of characters.

Generating Hidden Word Graphics

A frequent abuse of the Web is scripts that attempt to create dozens of email accounts or buy hundreds of concert tickets by parsing HTML forms and responding to them. To counteract this, many web sites now feature hidden word graphics that contain a word along with visual noise (in our case, lines drawn through the text) so that only a human can read the word. This ensures that only a human can use the form. We'll look at an XSLT extension function that, given a word, generates a hidden word graphic and a web page.

Given a secret word, our extension function creates a graphic containing that word and an HTML form that displays the graphic and asks the user to type the word hidden in the graphic. (Obviously, a complete solution would generate the server-side code, transient cookies, and other things to process the form, but that's beyond what we'll cover here. Our focus is on how to create the XSLT extension that creates the graphic.)

Our stylesheet looks like this:

```
<?xml version="1.0"?>
<!-- hidden-word-test.xsl -->
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:saxon="java:com.oreilly.xslt.HiddenWord"
  xmlns:xalan="xalan://com.oreilly.xslt.HiddenWord"
  xmlns:ora="http://www.oreilly.com/xslt"
  extension-element-prefixes="saxon xalan ora">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>
          Test of hidden word generator
        </title>
      </head>
      <body style="font-family: sans-serif;">
          <h1>Test of hidden word generator</h1>
        <xsl:comment>
          This <form> doesn't have a method or action
          attribute; a real form would, of course...
        </xsl:comment>
        <xsl:variable name="createGraphic">
          <xsl:choose>
            <xsl:when test="function-available('saxon:createJPEG')">
              <xsl:value-of select="saxon:createJPEG('hidden.jpg',
                                    'giraffe', 48, 200, 100)"/>
            </xsl:when>
```

```
            <xsl:when test="function-available('xalan:createJPEG')">
              <xsl:value-of select="xalan:createJPEG('hidden.jpg',
                                    'monkey', 48, 200, 100)"/>
            </xsl:when>
            <xsl:when test="function-available('ora:createJPEG')">
              <xsl:value-of select="ora:createJPEG('hidden.jpg',
                                   'okapi', 48, 200, 100)"/>
            </xsl:when>
            <xsl:otherwise>
              <xsl:value-of select="1"/>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:variable>
        <form align="center">
          <p>Enter the word hidden in the graphic below:</p>
          <xsl:choose>
            <xsl:when test="number($createGraphic)">
              <p style="font-style: italic; font-size: 150%;">
                <xsl:text>Sorry, the image function isn't available.</xsl:text>
              </p>
            </xsl:when>
            <xsl:otherwise>
              <p>
                <img src="hidden.jpg" width="200" height="100" border="3"/>
              </p>
            </xsl:otherwise>
          </xsl:choose>
          <p>
            <input type="text" name="hiddenWord" size="20" maxlength="20"/>
          </p>
          <p>
            <input type="submit" value="Submit"/>
          </p>
        </form>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

Our stylesheet uses the `function-available( )` function to check for each version of the extension function. If we're running the Saxon processor, the first `<xsl:when>` returns `true`, the second branch is `true` if we're running Xalan, and the third `<xsl:when>` is `true` if we're running the .NET processor. That assumes, of course, that the appropriate libraries are accessible. If the Java runtime can't find the class, `function-available( )` returns false.

Notice that the `<xsl:choose>` element is inside a `<xsl:variable>`. Our extension function returns `0` if the function is available and it works correctly. If none of the functions are avilable, the value of our variable is `1`; the functions themselves can return a nonzero return code if something goes wrong.

Putting this code inside a variable lets the extension functions return a value that isn't written to the output document. At the bottom of the stylesheet, the test `number($createGraphic)` converts the value returned by the extension function to a number. If the value isn't zero, we print an error message. Otherwise, we generate the form that contains the generated graphic.

Java Version

The two Java-based processors use the same code; the stylesheet simply uses the Saxon and Xalan conventions to identify the extension class. The extension function takes as its input the secret word, the name of the created JPEG file, the font size for the text, and the width and height of the JPEG file. Given those parameters, the code chooses a font at random and creates a blank canvas of the requested size. The Java code then checks to make sure the selected font exists (because Java runs on so many non-Windows platforms, we need to make sure). Once the font is set, we measure the dimensions of the secret word written in the font at the requested font size. If the word is too wide to fit into the graphic, we reduce the font size by 2 points until the word fits inside the canvas.

Once the font and font size are set, we draw the word on the canvas and finish by drawing random lines over the text. When we've drawn everything on the canvas, a couple of lines of Java code write the canvas out to a JPEG file. The last few lines of code here handle any exceptions that might occur. Here's the complete listing:

```
/*
 * HiddenWord.java
 */

package com.oreilly.xslt;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics2D;
import java.awt.GraphicsEnvironment;
import java.awt.Rectangle;
import java.awt.font.LineMetrics;
import java.awt.image.BufferedImage;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Random;

import com.sun.image.codec.jpeg.ImageFormatException;
import com.sun.image.codec.jpeg.JPEGCodec;
import com.sun.image.codec.jpeg.JPEGImageEncoder;

// This class creates a JPEG that contains a hidden word.
public class HiddenWord
```

```
{
  // Three fonts likely to installed on any system...
  // If you want to generate hidden words in a non-Latin language,
  // you'll need to change the font names.
  public static String[] fontNames = {"Arial", "Times", "Verdana"};

  // createJPEG() is the name of the extension function.
  public static int createJPEG(String secretWord,
                               String outputFilename,
                               Double dFontSize,
                               Double dWidth,
                               Double dHeight)
    throws IOException, FileNotFoundException,
      ImageFormatException, Exception
  {
    int rc = 0;

    try
    {
      int fontSize = dFontSize.intValue();
      int width = dWidth.intValue();
      int height = dHeight.intValue();

      // Create a new BufferedImage.  We'll use it as a canvas;
      // we draw the hidden word and the lines that obscure it
      // onto the canvas, then write it out to a JPEG file.
      BufferedImage bi =
        new BufferedImage(width, height,
                          BufferedImage.TYPE_3BYTE_BGR);

      Random r = new Random();
      String fontName = fontNames[r.nextInt(fontNames.length)];

      // Fill the new graphic with a white background
      Graphics2D g = bi.createGraphics();
      g.setColor(Color.WHITE);
      g.fill(new Rectangle(0, 0, width, height));
      g.setColor(Color.BLACK);

      int fontStyle = Font.BOLD;
      int textWidth = 0;
      int textHeight = 0;

      // Now we have to load the font.  There's a chance the
      // font we selected isn't available, so we look through
      // the list of fonts until we find the one we're looking for.
      // If the font we want isn't available, we use Arial.
      // This is much more complicated than the .NET version,
      // where we assume the fonts are installed on every Windows
      // system.
      GraphicsEnvironment ge = GraphicsEnvironment.
                               getLocalGraphicsEnvironment();
      Font allFonts[] = ge.getAllFonts();
      Font chosenFont = new Font("Arial", fontStyle, fontSize);
      g.setFont(chosenFont);
```

```
FontMetrics fm = g.getFontMetrics();

boolean fontNotFound = true;
int j = 0;
while (fontNotFound && (j < allFonts.length))
{
  if (allFonts[j].getFontName().contains(fontName))
  {
    chosenFont = allFonts[j].deriveFont(fontStyle, fontSize);
    if (!chosenFont.getFontName().equalsIgnoreCase(fontName))
    {
      fontStyle = Font.PLAIN;
      chosenFont = allFonts[j].deriveFont(fontStyle, fontSize);
    }
    g.setFont(chosenFont);
    fm = g.getFontMetrics();

    // We look at the width and height of the word as drawn in
    // the current font.  If it's too big to fit into the graphic
    // canvas, we reduce the font size and try it again.
    textWidth = fm.stringWidth(secretWord);
    while (textWidth > width && fontSize > 2)
    {
      fontSize -= 2;
      chosenFont = allFonts[j].deriveFont(fontStyle, fontSize);
      g.setFont(chosenFont);
      fm = g.getFontMetrics();
      textWidth = fm.stringWidth(secretWord);
    }
    if (fontSize < 1)
      chosenFont = allFonts[j].deriveFont(fontStyle, 12);

    g.setFont(chosenFont);
    fontNotFound = false;
  }
  else
    j++;
}

// Now we draw the string onto the canvas.  We use the dimensions of
// the canvas itself and the dimensions of the string to center the
// text in the graphic.
fm = g.getFontMetrics(chosenFont);
LineMetrics lm =
  chosenFont.getLineMetrics(secretWord, g.getFontRenderContext());
textHeight = (int)lm.getAscent();
textWidth = fm.stringWidth(secretWord);
g.drawString(secretWord, (width - textWidth) / 2,
             textHeight + 30);

// Now we'll draw some lines at random to obscure the text.
g.setStroke(new BasicStroke((float)2.0));

for (int i = 0; i < width / 30; i++)
  g.drawLine(0, r.nextInt(height), width, r.nextInt(height));
```

```
        int numLines = java.lang.Math.max(height, width) / 30;
        for (int i = 0; i < numLines; i++)
        {
          int nextX = r.nextInt(width);
          g.drawLine(nextX, 0, nextX, height);
          int nextY = r.nextInt(height);
          g.drawLine(0, nextY, width, nextY);
        }

        for (int i = 0; i < height / 20; i++)
          g.drawLine(r.nextInt(width), 0, r.nextInt(width), height);

        // We've drawn everything on the canvas that we wanted, so we'll
        // write the contents of the canvas out to a JPEG file.
        FileOutputStream fos = new FileOutputStream(outputFilename);
        JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(fos);
        encoder.encode(bi);
        fos.flush();
        fos.close();
      }
      catch (FileNotFoundException fnfe)
      {
        rc = 4;
        System.err.println(fnfe);
      }
      catch (IOException ioe)
      {
        rc = 8;
        System.err.println(ioe);
      }
      catch (Exception e)
      {
        rc = 12;
        System.err.println(e);
      }

      return rc;
    }
  }
```

The generated HTML form looks like Figure 9-5.

To illustrate how text sizing works, Figure 9-6 shows a hidden word graphic with a much longer word.

As you can see, the font size has been reduced so that the word fits inside the graphic.

.NET Version

The .NET version works similarly, although we need two C# files to use the extension. The first creates the `XslCompiledTransform` object and associates the extension function with it. The second file actually implements the extension, creating the hidden word graphic.

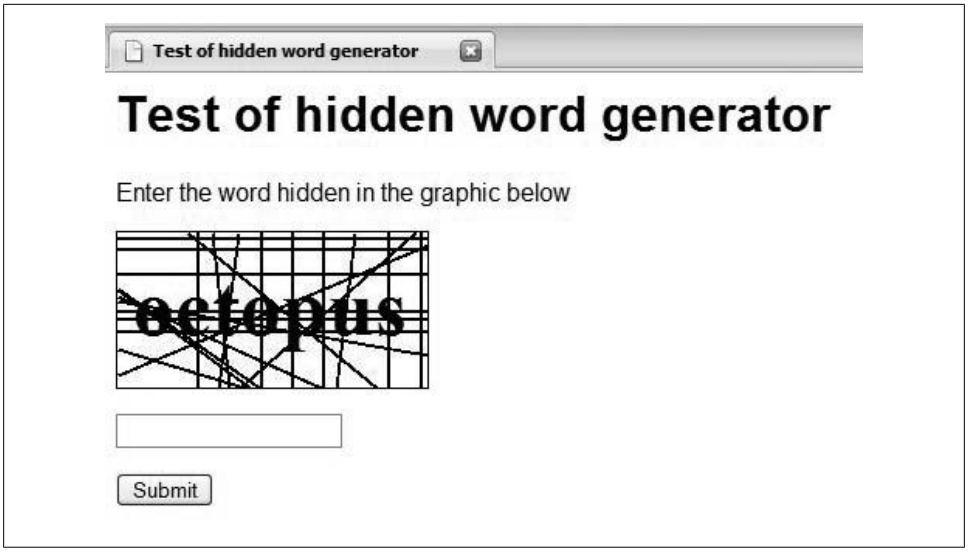


Figure 9-5. The HTML form with a hidden word graphic

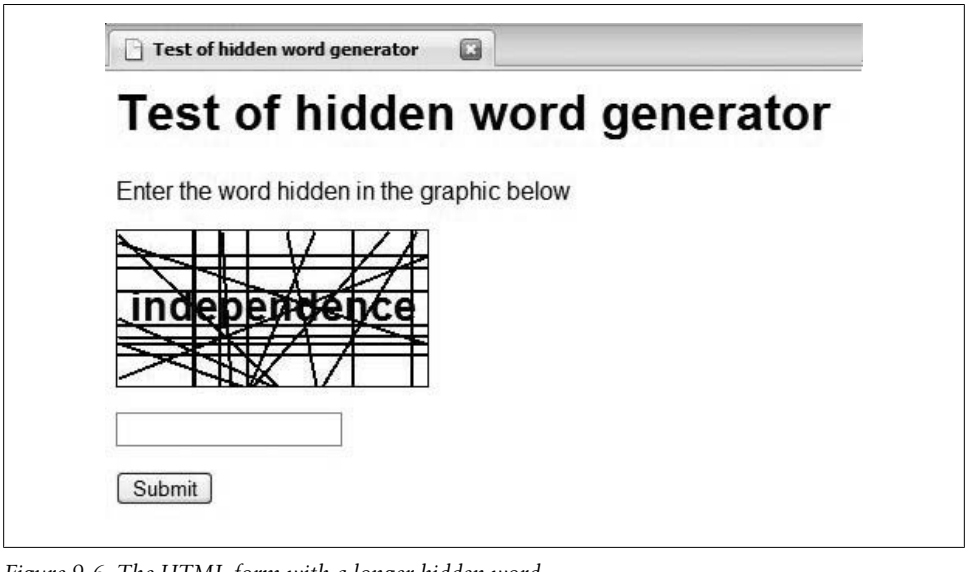

Test of hidden word generator
Test of hidden word generator
Enter the word hidden in the graphic below
independence
Submit

Figure 9-6. The HTML form with a longer hidden word

Here's the first C# file, which sets up the transformation object:

```
/*
 * Main.cs
 */

using System;
using System.Text;
```

```
using System.Xml;
using System.Xml.Xsl;

namespace com.oreilly.xslt
{
  class MainClass
  {
    static void Main(string[] args)
    {
      if (args.Length < 3)
      {
        System.Console.WriteLine("\nUsage: HiddenWordExample xml-file " +
          "xsl-file output-file");
        System.Console.WriteLine("\n  Example: HiddenWordExample " +
          "blank.xml hidden-word-test.xsl results.html");
      }
      else
      {
        // Create the stylesheet object and the XMLWriter that
        // writes the output to a file
        XslCompiledTransform stylesheet = new XslCompiledTransform();
        XmlTextWriter xWriter =
            new XmlTextWriter(args[2], Encoding.UTF8);

        // Use an XsltSettings object that allows executing scripts
        // (we need this for extensions), then load the stylesheet
        XsltSettings settings = new XsltSettings(true, true);
        stylesheet.Load(args[1], settings, new XmlUrlResolver());

        // Create an extension object
        HiddenWord hw = new HiddenWord();

        // We set up the extension function with an XsltArgumentList object.
        XsltArgumentList argList = new XsltArgumentList();
        argList.AddExtensionObject("http://www.oreilly.com/xslt", hw);

        // With everything in place, we call the Transform() method
        // to do the work...
        stylesheet.Transform(args[0], argList, xWriter);
      }
    }
  }
}
```

The C# file that creates the `XslCompiledTransform` has to define the object that implements the extension function (an instance of the `HiddenWord` class) and associate it with a namespace (`http://www.oreilly.com/xslt`). The namespace defined in the C# code must match the namespace defined in the stylesheet. When this code is built as an *.exe* file, it takes three arguments: the names of the input XML, the stylesheet, and the result files. If the user doesn't provide at least three arguments, we print an error message to avoid an `IndexOutOfRange` exception.

Now that we've set up the transformation, here's the actual code for the extension object. Although the methods and system objects are different from the Java version, the flow of the code is the same:

```
/*
 * HiddenWord.cs
 */

using System;
using System.Text;
using System.Xml;
using System.Xml.Xsl;
using System.Drawing;

// This class creates a JPEG that contains a hidden word.

namespace com.oreilly.xslt
{
  class HiddenWord
  {
    // Three fonts likely to be installed on a Windows system...
    // If you want to generate hidden words in a non-Latin language,
    // you'll need to change the font names.
    private static String[] fontNames = { "Arial", "Times", "Verdana" };

    // createJPEG() is the name of the extension function.
    public int createJPEG(String outputFilename,
                          String secretWord, Double dFontSize,
                          Double dWidth, Double dHeight)
    {
      int rc = 0;

      int fontSize = (int)dFontSize;
      int width = (int)dWidth;
      int height = (int)dHeight;

      Random r = new Random();

      String fontName = fontNames[r.Next(fontNames.Length)];

      // Create a new Bitmap.  We'll draw our text and graphics
      // onto the Bitmap, then write it out to a JPEG file.
      Bitmap pic = new Bitmap(width, height);
      Graphics context = Graphics.FromImage(pic);

      // First, fill the graphic with a white background
      SolidBrush brush = new SolidBrush(Color.White);
      context.FillRectangle(brush, 0, 0, width, height);

      // Load the randomly chosen font
      Font currentFont = new Font(fontName, fontSize, FontStyle.Bold);

      // Get the size of the word in the current font.  If it's too big
      // for the word to fit into the graphic, reduce the font size
      // until it fits.
```

```
        SizeF textSize = context.MeasureString(secretWord, currentFont);
        float textWidth = textSize.Width;

        while (textWidth > width && fontSize > 2)
        {
          fontSize -= 2;
          currentFont = new Font(fontName, fontSize, FontStyle.Bold);
          textSize = context.MeasureString(secretWord, currentFont);
          textWidth = textSize.Width;
        }
        if (fontSize < 1)
          currentFont = new Font(fontName, 12, FontStyle.Bold);

        float textHeight = textSize.Height;
        brush.Color = Color.Black;

        // Now draw the string onto the bitmap.  We center the text
        // by using the dimensions of the bitmap and the dimensions
        // of the string as drawn in the current font.
        context.DrawString(secretWord, currentFont, brush,
            (width - textWidth) / 2, (height - textHeight) / 2);

        // Now we draw some lines on the bitmap.  We generate the start
        // and endpoints of the lines at random.
        Pen pen = new Pen(Color.Black, (float)3);
        for (int i = 0; i < width / 30; i++)
          context.DrawLine(pen, 0, r.Next(height), width, r.Next(width));

        int numLines = Math.Max(height, width) / 30;
        for (int i = 0; i < numLines; i++)
        {
          int nextX = r.Next(width);
          context.DrawLine(pen, nextX, 0, nextX, height);
          int nextY = r.Next(height);
          context.DrawLine(pen, 0, nextY, width, nextY);
        }

        for (int i = 0; i < height / 20; i++)
          context.DrawLine(pen, r.Next(width), 0, r.Next(width), height);

        // Now we've drawn everything on the bitmap, so we write it out
        // to a file, using the JPEG format
        pic.Save(outputFilename, System.Drawing.Imaging.ImageFormat.Jpeg);

        return rc;
      }
    }
  }
```

As you can see, the C# code follows the same basic pattern as the Java code. We create a new canvas, reduce the font size until the given word fits inside the graphic, and then draw the text and the lines that obfuscate it and write it out to a JPEG file.

In this example, we demonstrated how to create JPEG graphics from text inside a stylesheet. We could have selected a secret word as a global parameter to the stylesheet,

or we could have selected something from an XML input document. As we mentioned, a complete solution would generate the server-side code to validate the word typed in by the user. Another thing to keep in mind is that the fonts we used here won't work for secret words written in non-Latin languages (Russian, Japanese, Greek, Chinese, Korean, and so forth). You could change the font names inside the code, or you could modify the stylesheet and the extension function to pass the font name as an argument from the stylesheet.

Example: Generating an SVG Pie Chart

As we outlined the functions and operators available in XPath and XSLT, you probably noticed that the mathematical functions at your disposal are rather limited, even in XSLT 2.0. In this example, we'll write an extension that provides a variety of trigonometric functions. We'll do this in several ways:

- We'll use the extension mechanisms in Xalan and Saxon to call static methods in the `java.lang.Math` class.
- We'll use the trigonometric functions in the EXSLT `math` library (more on EXSLT later).
- We'll use the Bean Scripting Framework, an interesting piece of code from the Apache Jakarta project that lets us write extension functions in a variety of scripting languages, including JRuby, JavaScript, Jython, and Jacl.
- We'll use classes from the .NET library to extend the Microsoft XSLT processor.

We'll start this section by building the stylesheet once, and then we'll discuss what's different in each iteration. Our scenario is that we want to generate a Scalable Vector Graphics (SVG) pie chart from an XML document. This document contains the sales figures for different stores of a company; we need to calculate the dimensions of the various slices of the pie graph for our SVG document. Here's the XML source we'll be working with:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate-sales.xml -->
<report month="8" year="2006">
  <caption>
    <heading>Chocolate bar sales</heading>
    <subheading>(units)</subheading>
  </caption>
  <store>
    <name>Carrboro</name>
    <brand name="Lindt">27408</brand>
    <brand name="Callebaut">8203</brand>
    <brand name="Valrhona">22101</brand>
    <brand name="Perugina">14336</brand>
    <brand name="Ghirardelli">19268</brand>
  </store>
  <store>
```

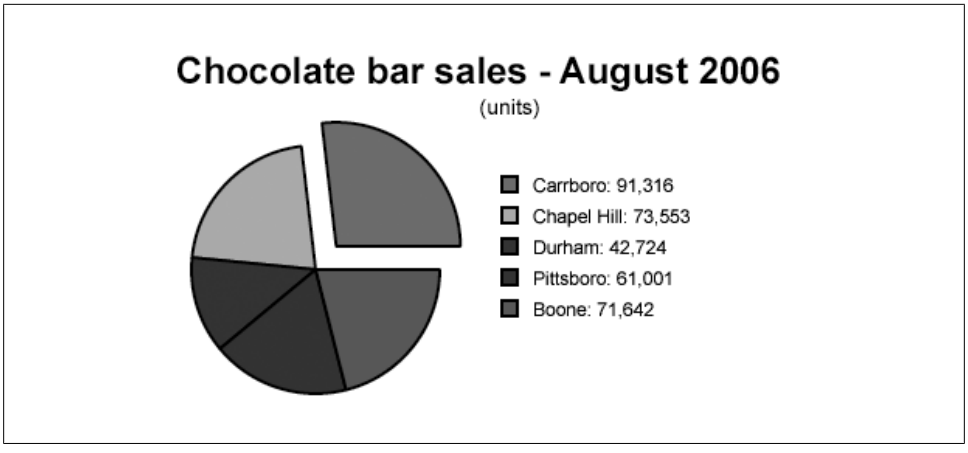


Figure 9-7. Our SVG pie chart

```
      <name>Chapel Hill</name>
      <brand name="Lindt">28503</brand>
      <brand name="Valrhona">7287</brand>
      <brand name="Perugina">12077</brand>
      <brand name="Ghirardelli">8392</brand>
      <brand name="Callebaut">17294</brand>
    </store>

...

  </store>
</report>
```

Our goal is to create an SVG file that looks like Figure 9-7.

To make our pie chart really useful, we'll take advantage of the scripting support available in the Adobe SVG Viewer. We'll generate the graphics of the chart, as well as EcmaScript functions to display additional detail. For example, moving the mouse over a slice of the pie shows the details for a particular store, as shown in Figure 9-8.

We'll take advantage of SVG's functions for line joins and other details that let our chart look as professional as possible. For example, we can zoom in on the graph, and the shapes and lines and text still look sharp, as shown in Figure 9-9.

The EcmaScript code we generate here is compatible with the Adobe SVG Viewer only; currently, Firefox displays the SVG pie chart, but it doesn't support any of the interactivity provided by the EcmaScript code.

XPath's limited math functions won't allow us to calculate the dimensions of the various arcs that make up the pie chart, so we'll use extension functions to solve this

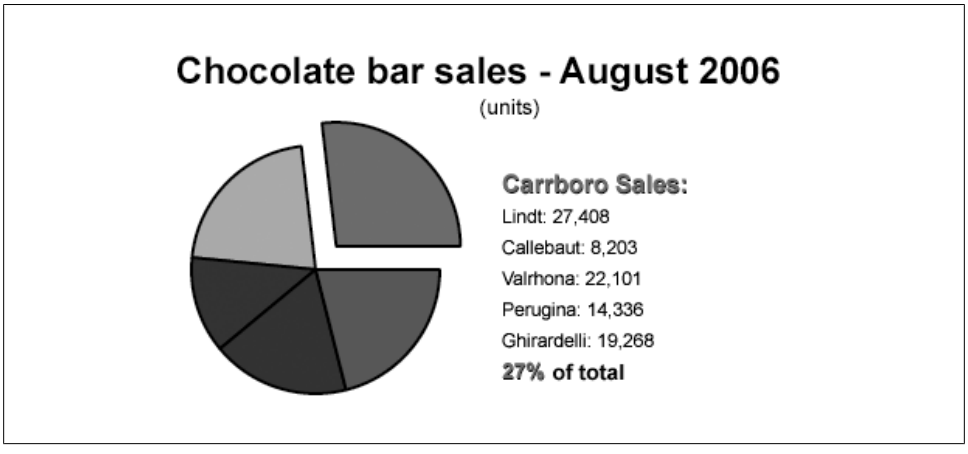


Figure 9-8. SVG chart changes in response to mouse events

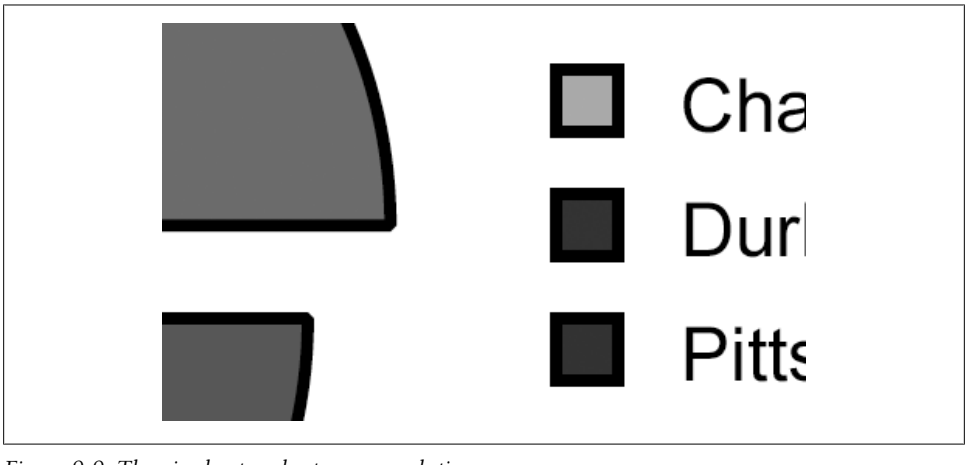


Figure 9-9. The pie chart scales to any resolution

problem. Fortunately for us, Java provides all the basic trigonometric functions we need in the `java.lang.Math` class. Even better, Xalan and Saxon make it easy for us to load this class and execute the static methods we need (`sin( )`, `cos( )` and `toRadians( )`). For the .NET case, we'll add a CDATA section containing EcmaScript code that the .NET XSLT processor can interpret.

EcmaScript is the name for the version of JavaScript standardized by the European Computer Manufacturers Association. We'll use the term "EcmaScript" here because the examples in the SVG spec use `<script type="text/ecmascript">` (although most people use the terms "EcmaScript" and "JavaScript" interchangeably).

We'll go over the relevant details as they appear in the stylesheet. First, we have to declare our namespace prefixes, just as we did when we used an extension element:

```
<?xml version="1.0"?>
<!-- piechart.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  extension-element-prefixes="xalan-java saxon-java msxsl ora months"
  xmlns:svg="http://www.w3.org/2000/svg"
  xmlns:xalan-java="http://xml.apache.org/xslt/java"
  xmlns:saxon-java="java:java.lang.Math"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:ora="http://www.oreilly.com/xslt"
  xmlns:months="http://www.oreilly.com/xslt/months">
```

We associated the `xalan-java` namespace prefix with the string `http://xml.apache.org/xslt/java`; Xalan uses this string to load Java classes and use their static methods. For Saxon support, we associated the `saxon-java` prefix with `java:java.lang.Math`. Notice that the two processors use different methods to load Java classes. The XSLT spec doesn't define any rules for how the namespace URIs are used to identify code that should be loaded by the processor, so each processor is likely to be different.

We also defined the `msxsl` and `ora` namespaces. The `msxsl` namespace lets us add EcmaScript directly to the stylesheet; we'll use the `ora` namespace to access that code. Finally, we define the `months` namespace. The stylesheet will contain a number of `<months:month>` elements, each of which defines the English name of a month of the year.

Next we define the output method and a couple of global variables:

```
<xsl:output method="xml"/>

<xsl:variable name="totalSales" select="sum(//brand)"/>
<xsl:variable name="stores" select="count(/report/store)"/>
```

Because SVG is an XML vocabulary, our output method is `xml`. The variable `$totalSales` is the total of all sales of all brands of chocolate in all stores, while the variable `$stores` is the number of stores in the report. We could recalculate these values every time we need them, but it's simpler to store them in global variables.

Next we define the `match="/"` template. This defines the `<svg>` element that contains our entire SVG document. Here's what the start of the template looks like:

```
<svg:svg width="450" height="300" version="1.1"
  baseProfile="full"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:ev="http://www.w3.org/2001/xml-events">
```

We're defining the dimensions of the SVG graphic and the version of SVG we're using, along with some namespaces the SVG interpreter needs. Now it's time to generate the EcmaScript functions that make our pie chart interactive. The original display of the pie chart shows the slices of the pie and a legend. Each slice of the pie is in a different

color, and the legend identifies the store associated with each piece of the pie and the total sales at each store. In addition, a pie slice for the first store in the XML source document is detached from the rest of the pie chart (this is sometimes called an "exploded" slice).

The EcmaScript code uses two state variables. The first is a boolean variable (`detailsViewOn`) that indicates whether the legend area is displaying the default legend or the detailed sales numbers for a particular store. The second is an integer (`currentStore`) that represents the number of the store whose detailed sales figures are currently displayed. The way the pie chart responds to mouse events is determined by the values of these two variables.

Here's what the EcmaScript functions do:

- When the user moves the mouse over a particular slice of the pie:
  - —If the legend area of the chart *is not* permanently displaying the details for a particular store (`detailsViewOn = false`), the legend is replaced with the detailed sales numbers for the store associated with that slice of the pie.
  - —If the legend area of the chart *is* permanently displaying the details for a particular store (`detailsViewOn = true`), nothing happens.
- When the user moves the mouse out of a particular slice of the pie:
  - —If the legend area of the chart *is not* permanently displaying the details for a particular store (`detailsViewOn = false`), then the detailed sales numbers for the store associated with the slice of pie are replaced with the default legend.
  - —If the legend area of the chart *is* permanently displaying the details for a particular store (`detailsViewOff = true`), nothing happens.
- When the user clicks on a particular slice of the pie:
  - —If the legend area of the chart *is not* permanently displaying the details for a particular store (`detailsViewOn = false`), display the detailed sales numbers in place of the default legend, set `detailsViewOn = true` and set the value of the current store (`currentStore = x`, where `x` is the number of the current store).
  - —If the legend area of the chart *is* permanently displaying the details for a particular store (`detailsViewOn = true`) *and* the store number corresponding to the clicked-upon slice is the same as the current store, then set `detailsViewOn = false` and set the value of the current store to `0`.
  - —If the legend area of the chart *is* permanently displaying the details for a particular store (`detailsViewOn = true`), but the store number corresponding to the clicked-upon slice *is not the same* as the current slice, set the value of the current store to the new value and replace the legend area with the detailed sales numbers for the new current store.

To sum up the behavior of the pie chart, the user can mouse over a slice of the pie to see detailed sales figures for a particular store. The user can click on a slice of pie to see

the details for that store permanently. Once the details for a particular store are displayed permanently, the user can click on another slice of the pie to see that store's details permanently, or they can click on the current store to return to the original display. (I hope that's clear; it's easy to figure out how the code works by interacting with it.)

We'll define three functions (`mouse_over( )`, `mouse_out( )`, and `click( )`) that handle the mouse events we care about. Here they are:

```
  <svg:script type="text/ecmascript">
    <xsl:text disable-output-escaping="yes">

detailsViewOn = false;
currentStore = 0;

function mouse_over(selectedItem, totalItems)
{
  if (!detailsViewOn || selectedItem == currentStore)
  {
    for (i = 1; totalItems >= i; i++)
    {
      obj = svgDocument.getElementById("details" + i);
      obj.setAttributeNS(null, "visibility", "hidden");
      obj = svgDocument.getElementById("legend" + i);
      obj.setAttributeNS(null, "visibility", "hidden");
    }

    obj = svgDocument.getElementById("details" + selectedItem);
    obj.setAttributeNS(null, "visibility", "visible");
  }
}

function mouse_out(totalItems)
{
  if (!detailsViewOn)
  {
    for (i = 1; totalItems >= i; i++)
    {
      obj = svgDocument.getElementById("details" + i);
      obj.setAttributeNS(null, "visibility", "hidden");
      obj = svgDocument.getElementById("legend" + i);
      obj.setAttributeNS(null, "visibility", "visible");
    }
  }
}

function click(selectedItem, totalItems)
{
  if (selectedItem != currentStore)
  {
    currentStore = selectedItem;
    mouse_over(selectedItem, totalItems);
    detailsViewOn = true;
  }
```

```
    else
    {
      obj = svgDocument.getElementById("details" + selectedItem);
      if (obj.getAttributeNS(null, "visibility") == "visible")
      {
        detailsViewOn = false;
        currentStore = 0;
      }
      else
      {
        currentStore = selectedItem;
        mouse_over(selectedItem, totalItems);
        detailsViewOn = true;
      }
    }
}
      //      </xsl:text></svg:script>
    <xsl:text>&#xA;</xsl:text>
```

Notice that we used `disable-output-escaping` so we don't have to worry about greater-than or less-than signs tripping up the XML parser.

If you're writing script code in an HTML file, you should write it inside a comment, allowing browsers that don't support scripting to safely ignore the code. The end of the script code should look like this:

```
...
      else
      {
        currentItem = selectedItem;
        mouse_over(selectedItem, totalItems);
        detailsViewOn = true;
      }
    }
  }
  //      -->
```

The double slash at the end of the script is a comment, which tells the scripting engine to ignore the `-->` at the end of the `<xsl:comment>` element. Without this slash, some processors attempt to process the end of the comment and issue an error message; after all, the `--` looks like the decrement operator. Keep this in mind when you're generating script code with your stylesheets; if you don't, you'll have trouble tracking down the occasional errors that occur in some browsers.

One more thing to note about generating EcmaScript code: in some cases you may want to put the code in a separate file. SVG allows you to put the EcmaScript code in a separate file and reference it from the SVG. (We did this in the first edition of this book.) Given the current state of the Adobe SVG Viewer, it's much simpler to put the EcmaScript code in the same file.

Now that we've created the EcmaScript code we need, we create the title and subtitle for the pie chart:

```
<!-- Generate the title and subtitle.  We draw the title and -->
<!-- subtitle on the chart, and we set the <svg:title> element -->
<!-- in the SVG document itself. -->
<xsl:variable name="titleText">
  <xsl:value-of select="/report/caption/heading"/>
  <xsl:text> - </xsl:text>
  <xsl:choose>
    <xsl:when test="function-available('math:cos')">
      <xsl:variable name="dateString">
        <xsl:text>--</xsl:text>
        <xsl:if test="string-length(/report/@month) = 1">
          <xsl:text>0</xsl:text>
        </xsl:if>
        <xsl:value-of select="/report/@month"/>
        <xsl:text>--</xsl:text>
      </xsl:variable>
      <xsl:value-of
        select="dates-and-times:month-name($dateString)"/>
      <xsl:text> </xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="/report/@month"/>
      <xsl:text>/</xsl:text>
    </xsl:otherwise>
  </xsl:choose>
  <xsl:value-of select="/report/@year"/>
</xsl:variable>

<svg:title>
  <xsl:value-of select="$titleText"/>
</svg:title>
<svg:text font-size="24px" text-anchor="start"
  font-weight="bold" x="10" y="40">
  <xsl:value-of select="$titleText"/>
</svg:text>
<svg:text font-size="14px" text-anchor="middle" y="60" x="225">
  <xsl:value-of select="/report/caption/subheading"/>
</svg:text>
```

Notice that we draw the text on the SVG canvas and also create an `<svg:title>` element for the SVG document. When the SVG document is rendered in a browser, this is treated as the title for the document. It's analogous to the HTML `<title>` element.

Now we're finally ready to process the `<store>` elements. Each `<store>` element contains the sales details for a particular store. We'll process each of these elements three ways:

1. First we create the pie slice. We'll look at the details in a minute, but our primary task here is to calculate the dimensions of the slice and to add the appropriate `onmouseover`, `onmouseout`, and `onclick` attributes. We do this in the `<xsl:template match="store" mode="pie">` template.
2. Next we create this store's entry in the legend. The legend has a colored square filled with the color for the current store, its name, and its total sales. The legend

entry for each store is drawn 20 pixels below the previous store's entry. We do this in the `<xsl:template match="store" mode="legend">` template.

3. Finally we create the detailed sales figures for the current store. The details view includes the name of the store, the sales of each brand at that store, and the current store's sales as a percentage of the total sales. Everything in the details view is drawn with the attribute `visibility="hidden"`. The details for a particular store are hidden until a particular mouse event occurs. This is done in the `<xsl:template match="store" mode="details">` template.

We'll go through these steps in detail now:

1. First of all, we need to draw the pie slice. This is done with an `<svg:path>` element. We'll use the trigonometry functions to calculate the angles of the edges of the pie slices, and we'll use SVG to draw a smooth arc between the outer endpoints of the edges. SVG also handles the details of joining the lines and filling the slice with a particular color.
2. For each slice of the pie, we calculate certain values and pass them as parameters to the `mode="pie"` template. First, we determine the color of the slice and the total sales for this particular store. We use the `position( )` function and the `mod` operator to calculate the color, and we use the `sum( )` function to calculate the sales for this store.
3. If this is the first slice of the pie, we'll explode it. That means that the first slice will be offset from the rest of the pie. We will set the variable `$explode` as follows:

```
<xsl:variable name="explode" select="position()=1"/>
```

4. The last value we calculate is the total sales of all previous stores. When we draw each slice of the pie, we rotate the coordinate axis a certain number of degrees. The amount of the rotation depends on how much of the total sales have been drawn so far. In other words, if we've drawn exactly 50% of the total sales, we'll rotate the axis 180 degrees (50% of 360). Rotating the axis simplifies the trigonometry we have to do. To calculate our total sales, we use the `preceding-sibling` axis:

```
<xsl:with-param name="runningTotal"
  select="sum(preceding-sibling::store/brand)"/>
```

5. Once we've calculated all the variables we need, we invoke the template:

```
<xsl:apply-templates select="." mode="pie">
  <xsl:with-param name="color" select="$color"/>
  <xsl:with-param name="storeSales" select="$storeSales"/>
  <xsl:with-param name="totalSales" select="$totalSales"/>
  <xsl:with-param name="runningTotal"
    select="sum(preceding-sibling::store/brand)"/>
  <xsl:with-param name="stores" select="$stores"/>
  <xsl:with-param name="explode" select="$explode"/>
</xsl:apply-templates>
```

6. Inside the template itself, our first step is to calculate the angle in radians of the current slice of the pie. This is the first time we use one of our extension functions:

```
<xsl:variable name="currentAngle">
  <xsl:choose>
    <xsl:when
      test="function-available('xalan-java:java.lang.Math.toRadians')">
      <xsl:value-of
        select="xalan-java:java.lang.Math.toRadians(
                ($storeSales div $totalSales) * 360.0)"/>
    </xsl:when>
    <xsl:when
      test="function-available('saxon-java:toRadians')">
      <xsl:value-of
        select="saxon-java:toRadians(
                ($storeSales div $totalSales) * 360.0)"/>
    </xsl:when>
    <xsl:when test="function-available('ora:cos')">
      <xsl:value-of
        select="($storeSales div $totalSales) *
                6.28318530717958647692528676655 9"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:message terminate="yes">
        <xsl:text>Sorry, this stylesheet can't generate any </xsl:text>
        <xsl:text>useful results &#xA;  without trigonometric </xsl:text>
        <xsl:text>functions available. </xsl:text>
      </xsl:message>
    </xsl:otherwise>
  </xsl:choose>
</xsl:variable>
```

In calculating the value of the variable, we use the XSLT `function-available( )` function. If we're processing this stylesheet with Xalan, the functions in the `xalan-java` namespace will be available. If we're using Saxon, the functions in the `saxon-java` namespace will be available. If we're using the .NET processor, the functions in the `ora` namespace will be available. If none of these functions are available, we terminate the processor. We can't generate a pie chart without trigonometric functions, so we exit.

The `$currentAngle` variable stores the angle in radians of the current pie slice. In other words, if the current store has 25% of the total sales for the company, we convert 90 degrees into radians and use that value to draw the slice of the pie. (We have to use radians because that's what the Java `cos( )` and `sin( )` functions require.) We'll use this value later with the `cos( )` and `sin( )` functions.

7. Now we're finally ready to draw the pie slice. We'll do this with an SVG `<path>` element. Here's what one looks like; we'll discuss what the attributes mean in a minute:

```
<svg:path
  fill="blue"
  stroke="black"
  stroke-width="2px"
  fillrule="evenodd"
  stroke-linejoin="bevel"
```

```
onmouseout="mouse_out(5);"
onmouseover="mouse_over(3, 5);"
onclick="click(3, 5);"
transform="translate(100,160)  rotate(-219.65188868902763)"
d="M 80 0 A 80 80 0 0 0 34.3848056509693 -72.23354581041326 L 0 0 Z "/>
```

This intimidating element was generated by this equally intimidating stylesheet fragment:

```
<svg:path fill="{$color}" stroke="black" stroke-width="2px"
  fillrule="evenodd" stroke-linejoin="bevel">
  <xsl:attribute name="onmouseout">
    <xsl:text>mouse_out(</xsl:text>
    <xsl:value-of select="$stores"/>
    <xsl:text>);</xsl:text>
  </xsl:attribute>
  <xsl:attribute name="transform">
    <xsl:choose>
      <xsl:when test="$explode">
        <xsl:text>translate(</xsl:text>
        <xsl:choose>
          <xsl:when
            test="function-available('xalan-java:java.lang.Math.cos')">
            <xsl:value-of
              select="(xalan-java:java.lang.Math.cos(
                      $currentAngle div 2) * 20) + 100"/>
            <xsl:text>,</xsl:text>
            <xsl:value-of
              select="(xalan-java:java.lang.Math.sin($currentAngle div 2)
                      * -20) + 160"/>
          </xsl:when>
          <xsl:when
            test="function-available('saxon-java:cos')">
            <xsl:value-of
              select="(saxon-java:cos($currentAngle div 2) * 20) + 100"/>
            <xsl:text>,</xsl:text>
            <xsl:value-of
              select="(saxon-java:sin($currentAngle div 2) * -20) + 160"/>
          </xsl:when>
          <xsl:when test="function-available('ora:cos')">
            <xsl:value-of
              select="(ora:cos($currentAngle div 2) * 20) + 100"/>
            <xsl:text>,</xsl:text>
            <xsl:value-of
              select="(ora:sin($currentAngle div 2) * -20) + 160"/>
          </xsl:when>
        </xsl:choose>
        <xsl:text>) </xsl:text>
      </xsl:when>
      <xsl:otherwise>
        <xsl:text>translate(100,160) </xsl:text>
      </xsl:otherwise>
    </xsl:choose>
    <xsl:text> rotate(</xsl:text>
    <xsl:value-of select="-1 * (($runningTotal div $totalSales) * 360.0)"/>
    <xsl:text>)</xsl:text>
```

```
    </xsl:attribute>
    <xsl:attribute name="onmouseover">
      <xsl:text>mouse_over(</xsl:text>
      <xsl:value-of select="$position"/>
      <xsl:text>, </xsl:text>
      <xsl:value-of select="$stores"/>
      <xsl:text>);</xsl:text>
    </xsl:attribute>
    <xsl:attribute name="onclick">
      <xsl:text>click(</xsl:text>
      <xsl:value-of select="$position"/>
      <xsl:text>, </xsl:text>
      <xsl:value-of select="$stores"/>
      <xsl:text>);</xsl:text>
    </xsl:attribute>
    <xsl:attribute name="d">
      <xsl:text>M 80 0 A 80 80 0 </xsl:text>
      <xsl:choose>
        <xsl:when test="$currentAngle > 3.14">
          <xsl:text>1 </xsl:text>
        </xsl:when>
        <xsl:otherwise>
          <xsl:text>0 </xsl:text>
        </xsl:otherwise>
      </xsl:choose>
      <xsl:text>0 </xsl:text>
      <xsl:choose>
        <xsl:when test="function-available('xalan-java:java.lang.Math.cos')">
          <xsl:value-of
            select="xalan-java:java.lang.Math.cos($currentAngle) * 80"/>
          <xsl:text> </xsl:text>
          <xsl:value-of
            select="xalan-java:java.lang.Math.sin($currentAngle) * -80"/>
        </xsl:when>
        <xsl:when test="function-available('saxon-java:cos')">
          <xsl:value-of select="saxon-java:cos($currentAngle) * 80"/>
          <xsl:text> </xsl:text>
          <xsl:value-of select="saxon-java:sin($currentAngle) * -80"/>
        </xsl:when>
        <xsl:when test="function-available('ora:cos')">
          <xsl:value-of
            select="ora:cos($currentAngle) * 80"/>
          <xsl:text> </xsl:text>
          <xsl:value-of
            select="ora:sin($currentAngle) * -80"/>
        </xsl:when>
      </xsl:choose>
      <xsl:text> L 0 0 Z </xsl:text>
    </xsl:attribute>
  </svg:path>
```

We'll cover the attributes of the SVG element in order. The `fill` attribute defines what color to use when filling the path. `stroke` defines the color of the line used to draw the path, and `stroke-width` defines the width of that line. `fillrule` and `stroke-linejoin` define details of how a path is filled and how lines are joined.

Next we generate the calls to our script functions. If this is the third slice (out of five) of the pie, calls to the script functions look like this:

```
onmouseout="mouse_out(5);"
onmouseover="mouse_over(3, 5);"
onclick="click(3, 5);"
```

The `mouse_out` function takes as its argument the number of slices of pie. It makes all of the details view elements hidden, and it makes the legend elements visible. (Those elements have `id`s of `legend1`, `details1`, and so forth.) `mouse_over` takes the value of the current slice of pie (the one that has the mouse over it) and the number of slices of pie. The `click` function also takes the value of the current slice of pie and the number of slices.

The real work begins with the `transform` attribute. It contains two operators, `translate` and `rotate`. What we're doing with the `transform` attribute is changing the position of the point `(0,0)` and the direction of the X and Y axes. We'll move and rotate the origin so the first side of the slice of the pie is along the X axis. (Trust me, it simplifies the math.)

That brings us to the gloriously cryptic `d` attribute. This attribute contains a number of drawing commands; in our previous example, we move the current point to `(80,0)` (`M` stands for move), and then we draw an elliptical arc (`A` stands for arc) with various properties. Finally, we draw a line (`L` stands for line) from the current point (the end of our arc) to the origin, and then we use the `Z` command, which closes the path by drawing a line from wherever we are to wherever we started.

If you really must know what the properties of the `A` command are, they are the two radii of the ellipse, the degrees by which the x-axis should be rotated, two parameters called the large-arc-flag and the sweep-flag that determine how the arc is drawn, and the x- and y-coordinates of the end of the arc. In our example here, the two radii of the ellipse are the same (we want the pie to be round, not elliptical). Next is the x-axis rotation, which is `0`. After that is the large-arc-flag, which is `1` if this particular slice of the pie is greater than 180 degrees: it's `0` otherwise. The sweep-flag is `0`, and the last two parameters, the x- and y-coordinates of the end point, are calculated. See the SVG specification for more details on the `path` and `shape` elements.

8. Our next task is to draw all of the legends. We'll create a separate legend for each slice of the pie. Initially, all of the separate legends will be invisible (`<g style="visibility:hidden">`, in SVG parlance); as we mouse over the various slices of the pie, different legends will become visible or invisible.

   When we apply our template, we pass in several parameters, including the color of the box in the legend entry and the y-coordinate offset where the legend entry should be drawn. We call this template once for each `<store>` element, ensuring that our legend identifies each slice of the pie, regardless of how many slices there are. For each slice, we draw a box filled with the appropriate color and write the name of the store next to it. We increment the y-coordinate by 20 pixels for each

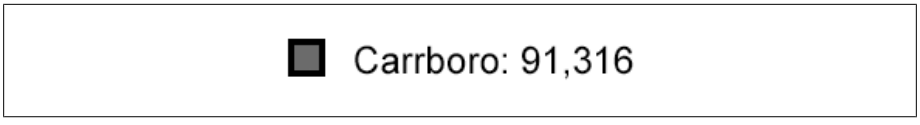


Figure 9-10. An item in the legend

item in the legend. To enable the EcmaScript functions we mentioned earlier, we give each item in the legend an ID: `legend1` for the first slice, `legend2` for the second, and so on.

Here's the template:

```
<xsl:template match="store" mode="legend">
  <xsl:param name="color" select="'red'"/>
  <xsl:param name="storeSales" select="'0'"/>
  <xsl:param name="y-legend-offset" select="'0'"/>
  <xsl:param name="position" select="'1'"/>

  <svg:g id="legend{$position}">

    <svg:text font-size="12px" text-anchor="start" x="240">
      <xsl:attribute name="y">
        <xsl:value-of select="$y-legend-offset"/>
      </xsl:attribute>
      <xsl:value-of select="name"/>
      <xsl:text>: </xsl:text>
      <xsl:value-of select="format-number($storeSales, '#,###.#')"/>
    </svg:text>

    <svg:path stroke="black" stroke-width="2px" fill="{$color}">
      <xsl:attribute name="d">
        <xsl:text>M 220 </xsl:text>
        <xsl:value-of select="$y-legend-offset - 10"/>
        <xsl:text> L 220 </xsl:text>
        <xsl:value-of select="$y-legend-offset"/>
        <xsl:text> L 230 </xsl:text>
        <xsl:value-of select="$y-legend-offset"/>
        <xsl:text> L 230 </xsl:text>
        <xsl:value-of select="$y-legend-offset - 10"/>
        <xsl:text> Z</xsl:text>
      </xsl:attribute>
    </svg:path>
  </svg:g>
</xsl:template>
```

An individual legend appears as in Figure 9-10.

A final note: we use the SVG group element (`<svg:g>`) here to group several items. Each group contains some text, and the colored box that indicates which slice of the pie represents each store in the company. By putting these into a group and giving that group a name, we can set the `visibility` property of the group and make everything it contains visible or invisible.

9. Our final task is to draw the details for each store of the company. The name of the store and this store's percentage of the total sales are drawn in the same color as its slice of the pie. For each brand sold by this store, we list the brand and its sales. (See Figure 9-11 to see exactly how this looks.) As we did with the template for drawing the legend, we put several items together in an SVG group. Here's the template:

```
<xsl:template match="store" mode="details">
  <xsl:param name="color"/>
  <xsl:param name="y-legend-offset"/>
  <xsl:param name="storeSales"/>
  <xsl:param name="totalSales"/>

  <svg:g visibility="hidden" id="details{$position}">
    <svg:text font-size="16px" font-weight="bold" text-anchor="start"
      fill="black" x="220.5" y="{$y-legend-offset + .5}">
      <xsl:value-of select="name"/><xsl:text> Sales:</xsl:text>
    </svg:text>
    <svg:text font-size="16px" font-weight="bold" text-anchor="start"
      fill="{$color}" x="220" y="{$y-legend-offset}">
      <xsl:value-of select="name"/><xsl:text> Sales:</xsl:text>
    </svg:text>
    <xsl:for-each select="brand">
      <svg:text font-size="12px" text-anchor="start" x="220"
        y="{$y-legend-offset + (position() * 20)}">
        <xsl:value-of select="@name"/>
        <xsl:text>: </xsl:text>
        <xsl:value-of select="format-number(., '#,###.#')"/>
      </svg:text>
    </xsl:for-each>
    <svg:text font-size="14px" font-weight="bold" text-anchor="start"
      fill="black" x="220.5"
      y="{$y-legend-offset + 20.5 + (count(brand) * 20)}">
      <xsl:value-of
        select="format-number($storeSales div $totalSales, '##%')"/>
      <xsl:text> of total</xsl:text>
    </svg:text>
    <svg:text font-size="14px" font-weight="bold" text-anchor="start"
      fill="{$color}" x="220"
      y="{$y-legend-offset + 20 + (count(brand) * 20)}">
      <xsl:value-of
        select="format-number($storeSales div $totalSales, '##%')"/>
    </svg:text>
  </svg:g>
</xsl:template>
```

Notice that we draw this item to be invisible (`visibility="hidden"`); we'll use our JavaScript effects to make the various legends and details visible or hidden. In our stylesheet, we draw the title of the current store using the same color we used for the slice of the pie, followed by the sales figures for each product sold in this store and the percentage of total sales. The details for each brand are drawn 20 pixels below the previous one, and the final line that indicates the percentage of sales delivered by this particular store is written 20 pixels below the final brand.

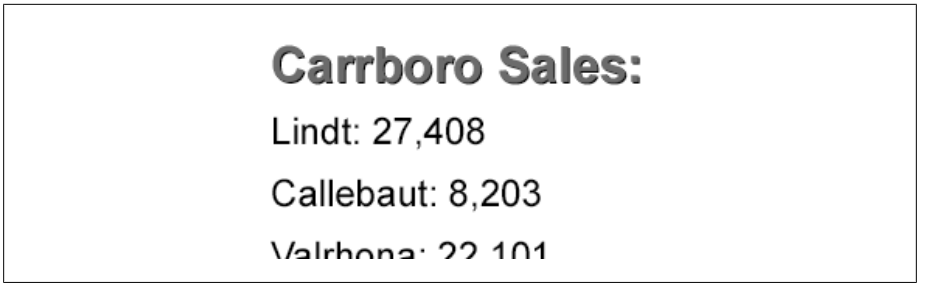


Figure 9-11. Sales details for a particular store

The sales details for a particular store appear as in Figure 9-11.

One subtle detail to note: we draw the name of the store and its percentage of sales twice, once in black and then once in the store's color. The black is .5 pixels down and to the right of the colored text in the foreground, which gives the display a nice shading effect.

Here's the complete stylesheet:

```
<?xml version="1.0"?>
<!-- piechart.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  extension-element-prefixes="xalan-java saxon-java msxsl ora months"
  xmlns:svg="http://www.w3.org/2000/svg"
  xmlns:xalan-java="http://xml.apache.org/xslt/java"
  xmlns:saxon-java="java:java.lang.Math"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:ora="http://www.oreilly.com/xslt"
  xmlns:months="http://www.oreilly.com/xslt/months">

  <months:month sequence="1">January</months:month>
  <months:month sequence="2">February</months:month>
  <months:month sequence="3">March</months:month>
  <months:month sequence="4">April</months:month>
  <months:month sequence="5">May</months:month>
  <months:month sequence="6">June</months:month>
  <months:month sequence="7">July</months:month>
  <months:month sequence="8">August</months:month>
  <months:month sequence="9">September</months:month>
  <months:month sequence="10">October</months:month>
  <months:month sequence="11">November</months:month>
  <months:month sequence="12">December</months:month>

  <msxsl:script implements-prefix="ora" language="C#">
    <![CDATA[
      public double cos(double d)
      {
        return Math.Round(Math.Cos(d), 4);
      }

      public double sin(double d)
```

```
    {
      return Math.Round(Math.Sin(d), 4);
    }
  ]]>
</msxsl:script>

<xsl:output method="xml"/>

<xsl:variable name="totalSales" select="sum(//brand)"/>
<xsl:variable name="stores" select="count(/sales/store)"/>

<xsl:template match="/">
  <svg:svg width="450" height="300" version="1.1"
    baseProfile="full"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:ev="http://www.w3.org/2001/xml-events">
    <xsl:comment>
      ECMAScript to handle mouse events
    </xsl:comment>

    <!-- Our generated SVG file has three functions: -->
    <!-- - mouse_over normally displays the details for a store -->
    <!--   in place of the legend. -->
    <!-- - mouse_out normally restores the legend when the mouse -->
    <!--   leaves a slice of the pie. -->
    <!-- - click handles a mouse click on a given slice of pie. -->
    <!--   If this is the first time we've clicked on this slice, -->
    <!--   this slice becomes the current item, and its sales -->
    <!--   details are displayed until the user clicks on another -->
    <!--   slice of pie. -->
    <!--   If this is the second time we've clicked on this slice, -->
    <!--   the details view is switched off and the normal mouse -->
    <!--   behavior returns. -->

    <svg:script type="text/ecmascript">
      <xsl:text disable-output-escaping="yes">

  detailsViewOn = false;
  currentItem = 0;

  function mouse_over(selectedItem, totalItems)
  {
    if (!detailsViewOn || selectedItem == currentItem)
    {
      for (i = 1; totalItems >= i; i++)
      {
        obj = svgDocument.getElementById("details" + i);
        obj.setAttributeNS(null, "visibility", "hidden");
        obj = svgDocument.getElementById("legend" + i);
        obj.setAttributeNS(null, "visibility", "hidden");
      }

      obj = svgDocument.getElementById("details" + selectedItem);
      obj.setAttributeNS(null, "visibility", "visible");
```

```
      }
    }

    function mouse_out(totalItems)
    {
      if (!detailsViewOn)
      {
        for (i = 1; totalItems >= i; i++)
        {
          obj = svgDocument.getElementById("details" + i);
          obj.setAttributeNS(null, "visibility", "hidden");
          obj = svgDocument.getElementById("legend" + i);
          obj.setAttributeNS(null, "visibility", "visible");
        }
      }
    }

    function click(selectedItem, totalItems)
    {
      if (selectedItem != currentItem)
      {
        currentItem = selectedItem;
        mouse_over(selectedItem, totalItems);
        detailsViewOn = true;
      }
      else
      {
        obj = svgDocument.getElementById("details" + selectedItem);
        if (obj.getAttributeNS(null, "visibility") == "visible")
        {
          detailsViewOn = false;
          currentItem = 0;
        }
        else
        {
          currentItem = selectedItem;
          mouse_over(selectedItem, totalItems);
          detailsViewOn = true;
        }
      }
    }
          //      </xsl:text></svg:script>
        <xsl:text>
</xsl:text>

        <!-- Generate the title and subtitle.  We draw the title and -->
        <!-- subtitle on the chart, and we set the <svg:title> element -->
        <!-- in the SVG document itself. -->
        <xsl:variable name="titleText">
          <xsl:value-of select="/report/caption/heading"/>
          <xsl:text> - </xsl:text>
          <xsl:value-of
            select="document('')/*/months:month
                    [@sequence=current()/report/@month]"/>
          <xsl:text> </xsl:text>
```

```
  <xsl:value-of select="/report/@year"/>
</xsl:variable>

<svg:title>
  <xsl:value-of select="$titleText"/>
</svg:title>
<svg:text font-size="24px" text-anchor="start"
  font-weight="bold" x="10" y="40">
  <xsl:value-of select="$titleText"/>
</svg:text>
<svg:text font-size="14px" text-anchor="middle" y="60" x="225">
  <xsl:value-of select="/report/caption/subheading"/>
</svg:text>

<xsl:variable name="totalSales" select="sum(//brand)"/>
<xsl:variable name="stores" select="count(/report/store)"/>

<xsl:for-each select="/report/store">
  <xsl:variable name="storeSales" select="sum(brand)"/>
  <xsl:variable name="color">
    <xsl:choose>
      <xsl:when test="(position() mod 6) = 1">
        <xsl:text>red</xsl:text>
      </xsl:when>
      <xsl:when test="(position() mod 6) = 2">
        <xsl:text>orange</xsl:text>
      </xsl:when>
      <xsl:when test="(position() mod 6) = 3">
        <xsl:text>purple</xsl:text>
      </xsl:when>
      <xsl:when test="(position() mod 6) = 4">
        <xsl:text>blue</xsl:text>
      </xsl:when>
      <xsl:when test="(position() mod 6) = 5">
        <xsl:text>green</xsl:text>
      </xsl:when>
      <xsl:otherwise>
        <xsl:text>gray</xsl:text>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  <xsl:variable name="explode" select="position()=1"/>

  <!-- Create the pie slice for each store -->
  <xsl:apply-templates select=".">
    <xsl:with-param name="color" select="$color"/>
    <xsl:with-param name="storeSales" select="$storeSales"/>
    <xsl:with-param name="totalSales" select="$totalSales"/>
    <xsl:with-param name="runningTotal"
      select="sum(preceding-sibling::store/brand)"/>
    <xsl:with-param name="stores" select="$stores"/>
    <xsl:with-param name="explode" select="$explode"/>
    <xsl:with-param name="position" select="position()"/>
  </xsl:apply-templates>
```

```
        <!-- Create the legend entry for each store -->
        <xsl:apply-templates select="." mode="legend">
          <xsl:with-param name="color" select="$color"/>
          <xsl:with-param name="storeSales" select="$storeSales"/>
          <xsl:with-param name="y-legend-offset"
            select="90 + (position() * 20)"/>
          <xsl:with-param name="position" select="position()"/>
        </xsl:apply-templates>

        <!-- Create the (initially hidden) sales details for -->
        <!-- each store -->
        <xsl:apply-templates select="." mode="details">
          <xsl:with-param name="color" select="$color"/>
          <xsl:with-param name="position" select="position()"/>
          <xsl:with-param name="y-legend-offset" select="110"/>
          <xsl:with-param name="storeSales" select="$storeSales"/>
          <xsl:with-param name="totalSales" select="$totalSales"/>
        </xsl:apply-templates>

      </xsl:for-each>
    </svg:svg>
</xsl:template>

<!-- For each store, create the pie slice (<svg:path>) and the -->
<!-- onmouseover(), onmouseout(), and onclick() event handlers. -->
<xsl:template match="store">
  <xsl:param name="color" select="'red'"/>
  <xsl:param name="runningTotal" select="'0'"/>
  <xsl:param name="totalSales" select="'0'"/>
  <xsl:param name="storeSales" select="'0'"/>
  <xsl:param name="stores" select="'5'"/>
  <xsl:param name="explode"/>
  <xsl:param name="position" select="'1'"/>

  <xsl:variable name="currentAngle">
    <xsl:choose>
      <xsl:when
        test="function-available('xalan-java:java.lang.Math.toRadians')">
        <xsl:value-of
          select="xalan-java:java.lang.Math.toRadians(
                  ($storeSales div $totalSales) * 360.0)"/>
      </xsl:when>
      <xsl:when
        test="function-available('saxon-java:toRadians')">
        <xsl:value-of
          select="saxon-java:toRadians(
                  ($storeSales div $totalSales) * 360.0)"/>
      </xsl:when>
      <xsl:when test="function-available('ora:cos')">
        <xsl:value-of
          select="($storeSales div $totalSales) *
                  6.28318530717958647692528676655 9"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:message terminate="yes">
```

```
          <xsl:text>Sorry, this stylesheet can't generate any </xsl:text>
          <xsl:text>useful results
without trigonometric </xsl:text>
          <xsl:text>functions available. </xsl:text>
        </xsl:message>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:variable>

  <svg:path fill="{$color}" stroke="black" stroke-width="2px"
    fillrule="evenodd" stroke-linejoin="bevel">
    <xsl:attribute name="onmouseout">
      <xsl:text>mouse_out(</xsl:text>
      <xsl:value-of select="$stores"/>
      <xsl:text>);</xsl:text>
    </xsl:attribute>
    <xsl:attribute name="transform">
      <xsl:choose>
        <xsl:when test="$explode">
          <xsl:text>translate(</xsl:text>
          <xsl:choose>
            <xsl:when
              test="function-available('xalan-java:java.lang.Math.cos')">
              <xsl:value-of
                select="(xalan-java:java.lang.Math.cos(
                        $currentAngle div 2) * 20) + 100"/>
              <xsl:text>,</xsl:text>
              <xsl:value-of
                select="(xalan-java:java.lang.Math.sin($currentAngle div 2)
                        * -20) + 160"/>
            </xsl:when>
            <xsl:when
              test="function-available('saxon-java:cos')">
              <xsl:value-of
                select="(saxon-java:cos($currentAngle div 2) * 20) + 100"/>
              <xsl:text>,</xsl:text>
              <xsl:value-of
                select="(saxon-java:sin($currentAngle div 2) * -20) + 160"/>
            </xsl:when>
            <xsl:when test="function-available('ora:cos')">
              <xsl:value-of
                select="(ora:cos($currentAngle div 2) * 20) + 100"/>
              <xsl:text>,</xsl:text>
              <xsl:value-of
                select="(ora:sin($currentAngle div 2) * -20) + 160"/>
            </xsl:when>
          </xsl:choose>
          <xsl:text>) </xsl:text>
        </xsl:when>
        <xsl:otherwise>
          <xsl:text>translate(100,160) </xsl:text>
        </xsl:otherwise>
      </xsl:choose>
      <xsl:text> rotate(</xsl:text>
      <xsl:value-of select="-1 * (($runningTotal div $totalSales) * 360.0)"/>
```

```
      <xsl:text>)</xsl:text>
    </xsl:attribute>
    <xsl:attribute name="onmouseover">
      <xsl:text>mouse_over(</xsl:text>
      <xsl:value-of select="$position"/>
      <xsl:text>, </xsl:text>
      <xsl:value-of select="$stores"/>
      <xsl:text>);</xsl:text>
    </xsl:attribute>
    <xsl:attribute name="onclick">
      <xsl:text>click(</xsl:text>
      <xsl:value-of select="$position"/>
      <xsl:text>, </xsl:text>
      <xsl:value-of select="$stores"/>
      <xsl:text>);</xsl:text>
    </xsl:attribute>
    <xsl:attribute name="d">
      <xsl:text>M 80 0 A 80 80 0 </xsl:text>
      <xsl:choose>
        <xsl:when test="$currentAngle > 3.14">
          <xsl:text>1 </xsl:text>
        </xsl:when>
        <xsl:otherwise>
          <xsl:text>0 </xsl:text>
        </xsl:otherwise>
      </xsl:choose>
      <xsl:text>0 </xsl:text>
      <xsl:choose>
        <xsl:when test="function-available('xalan-java:java.lang.Math.cos')">
          <xsl:value-of
            select="xalan-java:java.lang.Math.cos($currentAngle) * 80"/>
          <xsl:text> </xsl:text>
          <xsl:value-of
            select="xalan-java:java.lang.Math.sin($currentAngle) * -80"/>
        </xsl:when>
        <xsl:when test="function-available('saxon-java:cos')">
          <xsl:value-of select="saxon-java:cos($currentAngle) * 80"/>
          <xsl:text> </xsl:text>
          <xsl:value-of select="saxon-java:sin($currentAngle) * -80"/>
        </xsl:when>
        <xsl:when test="function-available('ora:cos')">
          <xsl:value-of
            select="ora:cos($currentAngle) * 80"/>
          <xsl:text> </xsl:text>
          <xsl:value-of
            select="ora:sin($currentAngle) * -80"/>
        </xsl:when>
      </xsl:choose>
      <xsl:text> L 0 0 Z </xsl:text>
    </xsl:attribute>
  </svg:path>
</xsl:template>

<!-- For each store, create an entry in the legend -->
<xsl:template match="store" mode="legend">
```

```
  <xsl:param name="color" select="'red'"/>
  <xsl:param name="storeSales" select="'0'"/>
  <xsl:param name="y-legend-offset" select="'0'"/>
  <xsl:param name="position" select="'1'"/>

  <svg:g id="legend{$position}">

    <svg:text font-size="12px" text-anchor="start" x="240">
      <xsl:attribute name="y">
        <xsl:value-of select="$y-legend-offset"/>
      </xsl:attribute>
      <xsl:value-of select="name"/>
      <xsl:text>: </xsl:text>
      <xsl:value-of select="format-number($storeSales, '#,###.#')"/>
    </svg:text>

    <svg:path stroke="black" stroke-width="2px" fill="{$color}">
      <xsl:attribute name="d">
        <xsl:text>M 220 </xsl:text>
        <xsl:value-of select="$y-legend-offset - 10"/>
        <xsl:text> L 220 </xsl:text>
        <xsl:value-of select="$y-legend-offset"/>
        <xsl:text> L 230 </xsl:text>
        <xsl:value-of select="$y-legend-offset"/>
        <xsl:text> L 230 </xsl:text>
        <xsl:value-of select="$y-legend-offset - 10"/>
        <xsl:text> Z</xsl:text>
      </xsl:attribute>
    </svg:path>
  </svg:g>
</xsl:template>

<!-- For each store, create the hidden list of sales details -->
<xsl:template match="store" mode="details">
  <xsl:param name="color" select="black"/>
  <xsl:param name="position" select="'0'"/>
  <xsl:param name="y-legend-offset"/>
  <xsl:param name="storeSales"/>
  <xsl:param name="totalSales"/>

  <svg:g visibility="hidden" id="details{$position}">
    <svg:text font-size="16px" font-weight="bold" text-anchor="start"
      fill="black" x="220.5">
      <xsl:attribute name="y">
        <xsl:value-of select="$y-legend-offset + .5"/>
      </xsl:attribute>
      <xsl:value-of select="name"/><xsl:text> Sales:</xsl:text>
    </svg:text>
    <svg:text font-size="16px" font-weight="bold" text-anchor="start"
      fill="{$color}" x="220">
      <xsl:attribute name="y">
        <xsl:value-of select="$y-legend-offset"/>
      </xsl:attribute>
      <xsl:value-of select="name"/><xsl:text> Sales:</xsl:text>
    </svg:text>
```

```
      <xsl:for-each select="brand">
        <svg:text font-size="12px" text-anchor="start" x="220">
          <xsl:attribute name="y">
            <xsl:value-of select="$y-legend-offset + (position() * 20)"/>
          </xsl:attribute>
          <xsl:value-of select="@name"/>
          <xsl:text>: </xsl:text>
          <xsl:value-of select="format-number(., '#,###.#')"/>
        </svg:text>
      </xsl:for-each>
      <svg:text font-size="14px" font-weight="bold" text-anchor="start"
        fill="black" x="220.5"
        y="{$y-legend-offset + 20.5 + (count(brand) * 20)}">
        <xsl:value-of select="format-number($storeSales div $totalSales, '##%')"/>
        <xsl:text> of total</xsl:text>
      </svg:text>
      <svg:text font-size="14px" font-weight="bold" text-anchor="start"
        fill="{$color}" x="220"
        y="{$y-legend-offset + 20 + (count(brand) * 20)}">
        <xsl:value-of select="format-number($storeSales div $totalSales, '##%')"/>

      </svg:text>
    </svg:g>
  </xsl:template>

</xsl:stylesheet>
```

In this example, we've used XSLT extension functions to add new capabilities to the XSLT processor. We needed a couple of simple trigonometric functions, and Saxon and Xalan's ability to use existing Java classes made adding new capabilities simple. You can use this technique to invoke static methods of Java classes anywhere you need them. Best of all, we didn't have to write any Java code to make this happen.

Writing Extensions in Other Languages

One of the nice features of Xalan-J's extension mechanism is that it supports the Bean Scripting Framework (BSF), an open source library from the Apache Software Foundation that allows you to execute code written in a variety of scripting languages. We'll take the SVG stylesheet we just discussed and implement it again, writing the extension functions in a variety of other languages. For our first example, we'll look at all the details of defining and invoking BSF extension functions; for the subsequent examples, we'll simply highlight the differences.

The Bean Scripting Framework supports many languages, including JRuby, Jython, Groovy, Jacl, NetRexx, PerlScript, Tcl, and VBScript. If you're using a Microsoft platform, BSF also supports Windows Script Technologies, so you may have even more choices if you're running some flavor of Windows.

Using the BSF requires two files on your `CLASSPATH`—*bsf.jar*, which is available at *http://jakarta.apache.org/bsf/index.html*, and *commons-logging-1.1.jar*, which is available at

http://jakarta.apache.org/commons/logging/—in addition to the JAR files for the language you're using. We'll point out those requirements as we go. Also be aware that system requirements can change; earlier versions of the BSF did not require the logging component.

Jython

We'll start our tour of BSF-supported languages with Jython, an implementation of Python written in Java. As you would expect, we must do several things to identify our extension code to Xalan. We'll cover them, and then look at the source of the extension functions. First we need to define the namespace prefixes we'll use:

```
<!-- piechart-jython.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:svg="http://www.w3.org/2000/svg"
  xmlns:months="http://www.oreilly.com/xslt/months"
  xmlns:lxslt="http://xml.apache.org/xslt"
  xmlns:jython-extension="http://www.jython.org"
  extension-element-prefixes="jython-extension"
  exclude-result-prefixes="lxslt">
```

We're generating SVG markup, so we need to define the `svg` namespace. The `months` namespace is for the `<months>` elements we use for a `document('')` lookup table (for variety's sake, we'll avoid using the EXSLT extensions here). For the last two namespaces, `lxslt` is the namespace Xalan uses to invoke the Bean Scripting Framework, and `jython-extension` is the prefix we use to identify the extension functions written in Jython.

In this example, we associated the `jython-extension` namespace prefix with the URI `http://www.jython.org`, which is the home page for the Jython project. The URI could be anything. The crucial identifier for the BSF is the name of the language, as we'll see in just a minute.

Our code goes in a `<lxslt:component>` element:

```
  <lxslt:component prefix="jython-extension"
    functions="cos sin">
    <lxslt:script lang="jython">
import math

def cos(d):
  return math.cos(d)

def sin(d):
  return math.sin(d)
    </lxslt:script>
  </lxslt:component>
```

The `prefix` attribute of the `<lxslt:component>` element associates the code with a namespace prefix, while the `functions` attribute lists the extension functions provided by this code. The `<lxslt:script>` element contains the actual code. *The* `lang` *attribute of the* `<lxslt:script>` *element is case-sensitive and must match the name of a language known to the implementation of the Bean Scripting Framework you're using*. Changing the `lang` attribute to `Jython` causes a fatal error.

Now that we've packaged and labeled our Jython code correctly, all we have to do is invoke it:

```
<xsl:value-of
  select="(jython-extension:cos(number($currentAngle div 2)) * 20) + 100"/>
```

Other than the `jython-extension` extension before the function call, the rest of our stylesheet is exactly the same. The `cos( )` and `sin( )` functions are part of the Python `math` library, so all we have to do was invoke them.

To use these extension functions, your `CLASSPATH` must contain *jython.jar*, available at *http://www.jython.org*, in addition to *bsf.jar* and *commons-logging-1.1.jar*.

We'll move on to other languages now, looking at the script and the `CLASSPATH` setup for each.

JRuby

JRuby is an implementation of the popular Ruby language written in Java. We define the namespace prefix `jruby-extension` and associate it with the URI `http://jruby.codehaus.org`. Here's the JRuby code:

```
<?xml version="1.0"?>
<!-- piechart-jruby.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:svg="http://www.w3.org/2000/svg"
  xmlns:months="http://www.oreilly.com/xslt/months"
  xmlns:lxslt="http://xml.apache.org/xslt"
  xmlns:jruby-extension="http://jruby.codehaus.org"
  extension-element-prefixes="jruby-extension"
  exclude-result-prefixes="lxslt">
...
  <lxslt:component prefix="jruby-extension"
    functions="cos sin">
    <lxslt:script lang="ruby">
      def cos(d)
        Math::cos(d)
      end

      def sin(d)
        Math::sin(d)
      end
    </lxslt:script>
  </lxslt:component>
```

To use these extension functions, your CLASSPATH must contain *jruby.jar*, available at *http://jruby.codehaus.org*, in addition to *bsf.jar* and *commons-logging-1.1.jar*.

The magic lang attribute value here is ruby. Using JRuby, jruby, or Ruby, all of which are reasonable choices, causes a runtime error.

JavaScript

We'll use Mozilla's Rhino JavaScript engine to illustrate JavaScript support. We define the namespace prefix javascript-extension and associate it with the URI http://www.mozilla.org/rhino. The JavaScript code looks like this:

```
<?xml version="1.0"?>
<!-- piechart-javascript.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:svg="http://www.w3.org/2000/svg"
  xmlns:months="http://www.oreilly.com/xslt/months"
  xmlns:lxslt="http://xml.apache.org/xslt"
  xmlns:javascript-extension="http://www.mozilla.org/rhino"
  extension-element-prefixes="javascript-extension"
  exclude-result-prefixes="lxslt">
...
  <lxslt:component prefix="javascript-extension"
    functions="cos sin">
    <lxslt:script lang="javascript">
      function cos(d)
      {
        return Math.cos(d);
      }

      function sin(d)
      {
        return Math.sin(d);
      }
    </lxslt:script>
  </lxslt:component>
```

To use these extension functions, your CLASSPATH must contain *js.jar*, available at *http://www.mozilla.org/rhino*, in addition to *bsf.jar* and *commons-logging-1.1.jar*.

Jacl

Jacl is a Tcl (Tool Command Language) interpreter written entirely in Java. We define the namespace prefix jacl-extension and associate it with the URI http://tcljava.sourceforge.net. The code is straightforward:

```
<?xml version="1.0"?>
<!-- piechart-jacl.xsl -->
```

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:svg="http://www.w3.org/2000/svg"
  xmlns:months="http://www.oreilly.com/xslt/months"
  xmlns:lxslt="http://xml.apache.org/xslt"
  xmlns:jacl-extension="http://tcljava.sourceforge.net"
  extension-element-prefixes="jacl-extension"
  exclude-result-prefixes="lxslt">
...
  <lxslt:component prefix="jacl-extension"
    functions="cos sin">
    <lxslt:script lang="jacl">
      proc cos {d}  {expr cos($d)}
      proc sin {d}  {expr sin($d)}
    </lxslt:script>
  </lxslt:component>
```

To use these extension functions, your `CLASSPATH` must contain *jacl.jar* and *tclja va.jar*, available at *http://tcljava.sourceforge.net*, in addition to *bsf.jar* and *commons-logging-1.1.jar*.

Using Extension Functions from the EXSLT Library

Earlier we mentioned the EXSLT project, an effort to define a common set of XSLT extension functions. For our next example, we'll use functions from the EXSLT library. Both Saxon and Xalan have EXSLT support built in, so it's very easy to use extensions from the EXSLT library.

Although a number of XSLT processors support EXSLT, be aware that EXSLT is implemented inconsistently between processors. Not all processors support all functions, and not all functions have the same signature or results.

EXSLT provides eight categories of functions:

Common
: Common functions for data typing and for working with node-sets. These are in the `http://exslt.org/common` namespace.

Dates and times
: Functions for manipulating dates and times. These are in the `http://exslt.org/dates-and-times` namespace.

Dynamic
: Functions to dynamically evaluate XPath expressions. These are in the `http://exslt.org/dynamic` namespace.

Functions
: Extension elements and functions that allow you to define your own functions. These are in the `http://exslt.org/functions` namespace.

Math
: Functions for trigonometry, exponentiation, logarithms, and other miscellaneous mathematical functions. These are in the `http://exslt.org/math` namespace.

Random
: A single function (`random-sequence( )`) that generates a sequence of random values between `0` and `1`. It is in the `http://exslt.org/random` namespace.

Regular expressions
: Functions that work with regular expressions. These are in the `http://exslt.org/regular-expressions` namespace.

Sets
: Functions to calculate the difference and intersection between sets. These are in the `http://exslt.org/sets` namespace.

Strings
: Functions for manipulating strings. These are in the `http://exslt.org/strings` namespace.

Many of the EXSLT functions and elements have been added as part of the language for XSLT 2.0. The EXSLT effort represents the most requested features missing in XSLT 1.0, so it's not surprising that EXSLT would have a strong impact on the design and features of XSLT 2.0.

You should also be aware that just because an XSLT processor supports EXSLT, it might not support all of the functions defined at `http://exslt.org`. There are also minor differences between the implementations, so beware.

We'll use EXSLT for three things: the `cos( )` and `sin( )` functions we used earlier, and the `month-name( )` function to get the name of the current month. Using `month-name( )` means we won't have to have the `<months:month>` elements we've used until now.

To keep this discussion short, we'll only look at the parts of the EXSLT stylesheet that are different from our original pie chart stylesheet. First of all, the `<xsl:stylesheet>` element has different namespace declarations, as you'd expect:

```
<?xml version="1.0"?>
<!-- piechart-exslt.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:svg="http://www.w3.org/2000/svg"
  xmlns:math="http://exslt.org/math"
  xmlns:dates-and-times="http://exslt.org/dates-and-times"
  extension-element-prefixes="dates-and-times math">
```

In our example, we're testing the availability of the math and date/time extension functions from EXSLT. The EXSLT functions are in two different namespaces, so we define the `dates-and-times` and `math` prefixes in addition to the ones we used earlier.

Using the `month-name( )` function takes some work. Here's how we use it:

```
<xsl:choose>
  <xsl:when test="function-available('math:cos')">
    <xsl:variable name="dateString">
      <xsl:text>--</xsl:text>
      <xsl:if test="string-length(/report/@month) = 1">
        <xsl:text>0</xsl:text>
      </xsl:if>
      <xsl:value-of select="/report/@month"/>
      <xsl:text>--</xsl:text>
    </xsl:variable>
    <xsl:value-of
      select="dates-and-times:month-name($dateString)"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="/report/@year"/>
  </xsl:when>
  <xsl:otherwise>
    <xsl:value-of select="/report/@month"/>
    <xsl:text>/</xsl:text>
    <xsl:value-of select="/report/@year"/>
  </xsl:otherwise>
</xsl:choose>
```

If the `dates-and-times:month-name( )` function is available, we use it to get the name of the month of the report. To get the name of the month, the implementations of the `month-name( )` function that ship with Xalan and Saxon require a string in the format `--MM--`. (That's the only format that works with both processors, at any rate. Each processor might work with other formats, but we'll keep our stylesheet simple by using one format for both processors.) We create a variable named `$dateString` that matches this format; if the value of the `month` attribute of the `<report>` element is one character long, we'll put a zero in front of it. If the EXSLT functions are available, `month-name('--08--')` returns `August`. If the functions aren't available, the stylesheet uses the `month` attribute directly and creates a heading such as `8/2006`.

Notice that the `function-available( )` call tests for the `math:cos` function. Calling `function-available('dates-and-times:month-name')` in Xalan-J 2.7.0 currently returns `false`, even though the function is available. Testing for the EXSLT function `math:cos` does return `true`, so that's what we use here; all that really matters is support for EXSLT.

Another difference is that EXSLT doesn't define a `toRadians( )` function, so we have to define that ourselves. Here's the difference between the EXLST version and the Saxon extension version:

```
<xsl:when test="function-available('math:cos')">
  <xsl:value-of
    select="($storeSales div $totalSales) *
```

```
            6.283185307179586476925286766559"/>
</xsl:when>
<xsl:when
  test="function-available('saxon-java:toRadians')">
  <xsl:value-of
    select="saxon-java:toRadians(
            ($storeSales div $totalSales) * 360.0)"/>
</xsl:when>
```

The rest of the stylesheet is basically the same. Here's how we generate the two coordinates for the SVG `translate( )` function, for example:

```
<xsl:when test="function-available('math:cos')">
  <xsl:value-of
    select="(math:cos($currentAngle div 2) * 20) + 100"/>
  <xsl:text>,</xsl:text>
  <xsl:value-of
    select="(math:sin($currentAngle div 2) * -20) + 160"/>
</xsl:when>
```

Accessing a Database with an Extension Element

The first edition of this book included an extension element that accessed an SQL database. Since that time, the Saxon and Xalan processors have added extension libraries that do this for us. We'll look at how to use those extensions here.

Our example here uses the open source Apache Derby database, available at *http://db.apache.org/derby/*. Here are the Derby commands and SQL statements that create and populate the database:

```
connect 'jdbc:derby://localhost:1527/books;create=true';
create schema doug;
set schema doug;
create table compbks (ISBN varchar(10) primary key, title varchar(50),
  author varchar(50), pages int, price double, publisher varchar(50));
insert into compbks values ('0596527217', 'XSLT', 'Doug Tidwell',
  800, 49.95, 'O''Reilly');
insert into compbks values ('0974152129', 'DocBook XSL: The Complete Guide',
  'Bob Strayton', 560, 49.95, 'Sagehill Enterprises');
insert into compbks values ('1565925807', 'DocBook: The Definitive Guide',
  'Norman Walsh and Leonard Muellner', 652, 39.95, 'O''Reilly');
insert into compbks values ('0596009747', 'XSLT Cookbook', 'Sal Mangano',
  751, 49.95, 'O''Reilly');
insert into compbks values ('0596003277', 'Learning XSLT',
  'Michael Fitzgerald', 352, 34.95, 'O''Reilly');
```

We create a database named `books` and a table named `compbks`, then we insert five rows into the table.

Saxon's SQL support requires a username that must match the database schema; that's why we create the database schema `doug` and associate it with the database. It also requires a password value, which must be at least one character; we'll look at those details in just a minute.

Accessing a Database in Saxon

The Saxon XSLT processor provides a set of extension elements that provide SQL functions. There are extension elements to connect and disconnect from a database, to run a database query, and to do updates, inserts, and deletes on database tables. To keep our example simple, we'll use `<sql:connection>` element to connect to a database, then we'll use `<sql:query>` to select items from the database and `<sql:close>` to close the connection.

Here's the complete stylesheet:

```
<?xml version="1.0"?>
<!-- saxon-sql.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sql="java:/net.sf.saxon.sql.SQLElementFactory"
  xmlns:saxon="http://saxon.sf.net/"
  extension-element-prefixes="saxon sql">

  <xsl:output method="html"/>

  <xsl:template match="/">

    <!-- Create the JDBC connection -->
    <xsl:variable name="connection"
      as="java:java.sql.Connection"
      xmlns:java="http://saxon.sf.net/java-type">
      <sql:connect database="jdbc:derby://localhost:1527/books"
        driver="org.apache.derby.jdbc.ClientDriver"
        user="doug"
        password="x"/>
    </xsl:variable>

    <!-- Run the query -->
    <xsl:variable name="queryResults">
      <sql:query connection="$connection" table="compbks"
        column="*" row-tag="tr" column-tag="td"/>
    </xsl:variable>

    <html>
      <head>
        <title>Computer books in our database</title>
      </head>
      <body style="font-family: sans-serif;">
        <h1>Computer books in our database</h1>
        <p>Here are the
        <xsl:value-of select="count($queryResults/tr)"/>
```

```
        computer books we have in stock:</p>
        <table border="1" cellpadding="5">
          <tr>
            <th>ISBN</th>
            <th>Title</th>
            <th>Author</th>
            <th>Pages</th>
            <th>List price</th>
            <th>Publisher</th>
          </tr>
          <xsl:copy-of select="$queryResults"/>
        </table>
      </body>
    </html>

    <!-- Close the connection -->
    <sql:close connection="$connection"/>
  </xsl:template>

</xsl:stylesheet>
```

We use the Saxon extension elements in three places. First, we create the database connection with the `<sql:connect>` element. We have to specify the URL of the database, the Java JDBC driver we use to access the database, and a username and password. Note that both the username and password must be at least one character long. Even if your database isn't password protected (the Derby database we use here isn't), you still have to provide a username and password. To complicate things further, the username must match the name of the database schema. That's why we created the database schema `doug` when we created the Derby database.

The second extension element we use is `<sql:query>`. This is where we actually run the query and get the result set. The variable `$queryResults` contains a tree of elements representing the result set. In our example here, we specify the name of the database table, the columns to return (our example here means `select *`, although we could be more specific if we wanted), and the elements used for the rows and columns in the results. We're generating an HTML table here, so we use `tr` and `td`.

After using the `<sql:query>` element, the variable `$queryResults` contains a tree of elements that represent the results of the database query. We use the `count( )` function to indicate how many books were found in the query, then we create our table. We use the HTML `<table>`, `<tr>`, and `<th>` elements to set up the table, then we copy the `$queryResults` variable to the output.

Once the HTML table is built, we use the `<sql:close>` element to close the database connection.

The results of our stylesheet appear as in Figure 9-12.

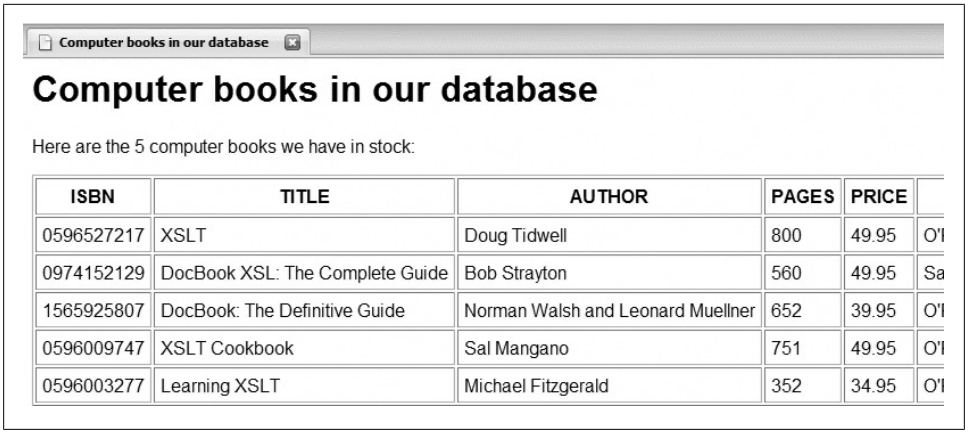


Figure 9-12. An HTML table built with Saxon's SQL extension elements

Be sure your CLASSPATH contains the classes for the Saxon SQL extension package and the JDBC driver classes; otherwise, your stylesheets won't work.

Saxon's SQL extension elements are built on JDBC, so they're extremely flexible. For example, to use this stylesheet with DB2, the only thing we have to change is the attributes of the `<sql:connect>` element:

```
<?xml version="1.0"?>
<!-- saxon-sql2.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sql="java:/net.sf.saxon.sql.SQLElementFactory"
  xmlns:saxon="http://saxon.sf.net/"
  extension-element-prefixes="saxon sql">

  ...

    <!-- Create the JDBC connection -->
    <xsl:variable name="connection"
      as="java:java.sql.Connection"
      xmlns:java="http://saxon.sf.net/java-type">
      <sql:connect database="jdbc:db2:books"
        driver="COM.ibm.db2.jdbc.app.DB2Driver"
        user="Skippy"
        password="xxxxxxxx"/>
    </xsl:variable>

  ...

</xsl:stylesheet>
```

The DB2 version of the database is password-protected, so we have to provide a username and password that are defined on the system. We changed one element, and now

our stylesheet is accessing a completely different data source. As with any database, we have to know the JDBC driver class and the URL format for the database, but that's the only change between the two stylesheets. The results of the new stylesheet, though, are exactly the same.

Be aware that the name of the JDBC driver class is case-sensitive. If you ask for the class `com.ibm.db2...` instead of `COM.ibm.db2...`, you'll get a runtime error.

Accessing a Database in Xalan

Xalan provides a set of extension functions that give us access to any JDBC data source. The extension functions we'll use are `sql:new( )` to create a new database connection, `sql:query( )` to execute a query, and `sql:close( )` to close the connection. In addition, we'll use the function `sql:disableStreamingMode( )` to get around a "feature" of the Xalan SQL library.

Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- xalan-sql.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sql="org.apache.xalan.lib.sql.XConnection"
  extension-element-prefixes="sql">

  <xsl:output method="html"/>

  <xsl:template match="/">

    <!-- Create the JDBC connection -->
    <xsl:variable name="books"
      select="sql:new('org.apache.derby.jdbc.ClientDriver',
              'jdbc:derby://localhost:1527/books')"/>

    <!-- Workaround for a bug in Xalan's SQL extension? -->
    <xsl:variable name="streaming"
      select="sql:disableStreamingMode($books)"/>

    <!-- Run the query -->
    <xsl:variable name="queryResults"
      select="sql:query($books, 'select * from doug.compbks')"/>

    <html>
      <head>
        <title>Computer books in our database</title>
      </head>
      <body style="font-family: sans-serif;">
        <h1>Computer books in our database</h1>
        <p>Here are the
        <xsl:value-of select="count($queryResults/sql/row-set/row)"/>
```

```
        computer books we have in stock:</p>
        <table border="1" cellpadding="5">
          <tr>
            <xsl:for-each select="$queryResults/sql/metadata/column-header">
              <th>
                <xsl:value-of select="@column-label"/>
              </th>
            </xsl:for-each>
          </tr>
          <xsl:apply-templates select="$queryResults/sql/row-set/row"/>
        </table>
      </body>
    </html>

    <!-- Close the connection -->
    <xsl:value-of select="sql:close($books)"/>
  </xsl:template>

  <xsl:template match="row">
    <tr>
      <xsl:apply-templates select="col"/>
    </tr>
  </xsl:template>

  <xsl:template match="col">
    <td>
      <xsl:value-of select="text()"/>
    </td>
  </xsl:template>

</xsl:stylesheet>
```

We start by creating a variable named `$books` that represents the connection to the database. Notice that the Xalan SQL library doesn't require us to provide a username and password; we simply specify the JDBC driver class name and the database URL. The `sql:disableStreamingMode($books)` call lets us access all of the results in the result set. Without this, the result set never has more than one result. Xalan-J 2.7.0 works this way, it's possible that future versions of the Xalan SQL library will work more sensibly.

Notice that the Xalan library gives us access to the information provided by the Java `ResultSetMetaData` class. In this case, we're using the column headers defined in our database, defined in the XPath hierarchy `sql/metadata/column-header`. Each column header is displayed in a `<th>` element inside the table. The Saxon SQL library doesn't provide this information, although it would be simple to extend the source code to do so.

Also notice that Xalan gives us more flexibility in the query statement. We could add an `ORDER BY` or `GROUP BY` clause to the statement if we want. The table name is specified differently here as well. Notice that we're asking for the table `doug.compbks`, which refers to the table named `compbks` that's in the `doug` database schema. If we simply ask for the

table compbks, the JDBC driver tells us that table doesn't exist. (Remember, we had to add the doug schema so the Saxon stylesheet would work.)

Once we've generated the headings for the columns in the table, it's time to process everything in the result set. The rows and columns from the result set are returned in a <row-set> element; each row is in a <row> element inside the <row-set>, and each column is in a <col> inside a <row>. The templates we define for the <row> and <col> elements are straightforward. The results are exactly the same as those for the stylesheets that used the Saxon SQL extensions.

To complete this short tour of the Xalan SQL extensions, we'll look at a second stylesheet that accesses a DB2 database. The differences, as you'd expect, are slight:

```
<?xml version="1.0"?>
<!-- xalan-sql2.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sql="org.apache.xalan.lib.sql.XConnection"
  extension-element-prefixes="sql">

  ...

    <!-- Create the JDBC connection -->
    <xsl:variable name="books"
      select="sql:new('COM.ibm.db2.jdbc.app.DB2Driver',
              'jdbc:db2:books', 'Skippy', 'xxxxxxxx')"/>

    ...

    <!-- Run the query -->
    <xsl:variable name="queryResults"
      select="sql:query($books, 'select * from compbks')"/>

  ...

</xsl:stylesheet>
```

We have to specify a different JDBC driver class and a database URL in a different format. This version of the sql:new( ) function accepts a username and password as two of its parameters. The other difference here is that we don't specify a database schema name for the database table. The default database schema in DB2 is the name of the user who created the database, so we can simply ask for the table compbks. If the table was created under another database schema, we would have to include the schema name here.

Creating a Photo Album with an Extension Element

As a final example, we'll create an extension element that generates an HTML table that displays all of the photographs in a given directory. The extension element takes three parameters: the directory, how many images are displayed on each row of the

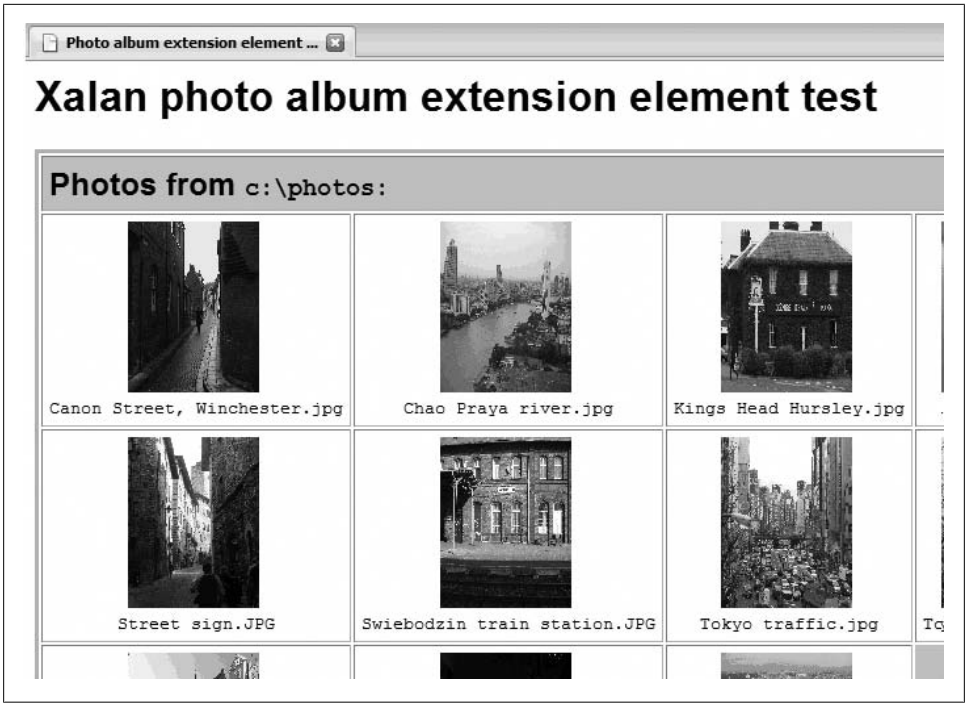


Figure 9-13. A photo album generated by an extension element

table, and whether the filename should be displayed under the photograph. The generated table displays a reduced version of each photograph; each small photo is a link to the actual file. The generated file looks like Figure 9-13.

We'll look at three versions of this extension here. The first two are written in Java, one version for Xalan and one version for Saxon. These two processors use different mechanisms to access the code; they also have different mechanisms to return the tree of HTML elements back to the stylesheet. The third version is written in C# for the .NET platform. The .NET version is written as an extension *function* rather than an element, although it still returns the tree of HTML elements we need. The HTML generated by the three implementations is the same.

All of the versions of our extension use this XML source file:

```
<?xml version="1.0"?>
<!-- photo-album.xml -->
<photo-album
  directory="c:\photos"
  imagesPerRow="5"
  includeFilenames="yes"/>
```

The XML source contains the three parameters we need for our extension: the directory that contains the images, the number of images per row, and whether to include the

filenames in the table. As we'll see, the extension code varies widely in the three examples, but the input data and the HTML output is the same.

The overall flow of the code is the same in all three extensions:

1. Get the attributes from the extension element. In some cases, we'll do that in the stylesheet; in others, the extension element will be able to get its own parameters.
2. See whether the requested directory exists. If it doesn't, create a table that includes a message such as "Directory doesn't exist."
3. Assuming the directory exists, see whether there are any images inside it. If it doesn't exist, create a table that includes a message such as "Directory doesn't contain any images."
4. Assuming the directory exists and contains some images, create a table with `<img>` elements that reference those images.

Xalan Java Version

We'll start with the Xalan Java version of the extension. The return type of the extension element is `org.w3c.dom.Element`. To start with, we'll look at the XSLT stylesheet that controls everything:

```
<?xml version="1.0"?>
<!-- xalan-photo-album.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xpa="xalan://com.oreilly.xslt.xalan.XalanPhotoAlbumExtension"
  extension-element-prefixes="xpa">

  <xsl:output method="html"/>

  <xsl:template match="photo-album">
    <html>
      <head>
        <title>Photo album extension element test</title>
      </head>
      <body style="font-family: sans-serif;">
        <xsl:choose>
          <xsl:when test="element-available('xpa:XalanPhotoAlbum')">
            <h1>Xalan photo album extension element test</h1>
            <xpa:XalanPhotoAlbum/>
          </xsl:when>
          <xsl:otherwise>
            <p>
              <i>[Sorry, the photo album function is not available.]</i>
            </p>
          </xsl:otherwise>
        </xsl:choose>
      </body>
    </html>
  </xsl:template>
```

```
</xsl:stylesheet>
```

Here's the code for the extension element:

```
/*
 * XalanPhotoAlbumExtension.java
 * Created on Nov 28, 2006 by Doug Tidwell
 */

package com.oreilly.xslt.xalan;

import java.io.File;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.apache.xalan.extensions.XSLProcessorContext;
import org.apache.xalan.templates.ElemExtensionCall;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

//This class creates a photo album with all of the images in a
//given directory.
public class XalanPhotoAlbumExtension
{
  // XalanPhotoAlbum is the name of the extension element.  Notice that
  // it returns an Element node; that node is inserted into the output tree.
  public static Element XalanPhotoAlbum(XSLProcessorContext context,
      ElemExtensionCall elem)
    throws ParserConfigurationException
  {
    Element contextNode = (Element) context.getContextNode();

    // For parameters, we get the name of the directory, the number of
    // images per row, and whether the names of the files should be displayed.

    int imagesPerRowValue;
    try
    {
      imagesPerRowValue =
        Integer.parseInt(contextNode.getAttribute("imagesPerRow"));
    }
    catch (NumberFormatException nfe)
    {
      imagesPerRowValue = 5;
    }

    String directoryName = contextNode.getAttribute("directory");

    boolean includeFilenamesFlag =
      (contextNode.getAttribute("includeFilenames").
          equalsIgnoreCase("yes"));

    // We're building a DOM tree, so we create a Document and some
```

```
// elements beforehand.  We'll return a <table> element that
// contains the images or an error message if something goes wrong.
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.newDocument();

Element a = null, br = null, img = null, span = null,
table = null, td = null, tr = null;

table = doc.createElement("table");
table.setAttribute("border", "3");
table.setAttribute("cellpadding", "5");

File dir = new File(directoryName);

// Return an error message if the directory doesn't exist
if (!dir.exists())
{
  tr = doc.createElement("tr");

  td = doc.createElement("td");
  td.setAttribute("style", "font-weight:bold; font-size: 150%");

  td.appendChild(doc.createTextNode("The directory "));

  span = doc.createElement("span");
  span.setAttribute("style", "font-family: monospace;");
  span.appendChild(doc.createTextNode(directoryName));
  td.appendChild(span);

  td.appendChild(doc.createTextNode(" does not exist!"));
  tr.appendChild(td);
  table.appendChild(tr);
}
else
{
  // Use the filename filter to find the images
  String[] graphicsFiles = dir.list(new GraphicsFilenameFilter());
  int numFiles = 0;
  if (graphicsFiles != null)
    numFiles = graphicsFiles.length;

  // Return an error message if the directory doesn't contain
  // any images
  if (numFiles == 0)
  {
    tr = doc.createElement("tr");

    td = doc.createElement("td");
    td.setAttribute("style", "font-weight:bold; font-size: 150%");

    td.appendChild(doc.createTextNode("The directory "));

    span = doc.createElement("span");
    span.setAttribute("style", "font-family: monospace;");
```

```
    span.appendChild(doc.createTextNode(directoryName));
    td.appendChild(span);

    td.appendChild(doc.createTextNode(" doesn't contain any images!"));
    tr.appendChild(td);
    table.appendChild(tr);
  }

  // We've got images, so let's process 'em...
  else
  {
    tr = doc.createElement("tr");

    td = doc.createElement("td");
    td.setAttribute("colspan", String.valueOf(imagesPerRowValue));
    td.setAttribute("style",
                    "font-weight:bold; background: #CCCCCC; " +
                    "font-size: 150%");
    td.appendChild(doc.createTextNode("Photos from "));

    span = doc.createElement("span");
    span.setAttribute("style", "font-family: monospace;");
    span.appendChild(doc.createTextNode(directoryName + ":"));
    td.appendChild(span);

    tr.appendChild(td);
    table.appendChild(tr);

    int filesProcessed = 0;
    boolean emptyColumnNotCreated = true;

    while (filesProcessed < numFiles)
    {
      tr = doc.createElement("tr");
      for (int i = 0; i < imagesPerRowValue; i++)
      {
        if (filesProcessed < numFiles)
        {
          td = doc.createElement("td");
          td.setAttribute("style", "text-align: center;");

          String qualifiedFilename = "file:\\\\" +
          directoryName + File.separatorChar +
          graphicsFiles[filesProcessed];

          a = doc.createElement("a");
          a.setAttribute("href", qualifiedFilename);

          img = doc.createElement("img");
          img.setAttribute("src", qualifiedFilename);
          img.setAttribute("width", "100px");
          img.setAttribute("height", "130px");
          img.setAttribute("border", "0");
          a.appendChild(img);
```

```
                td.appendChild(a);

                if (includeFilenamesFlag)
                {
                  br = doc.createElement("br");
                  td.appendChild(br);

                  span = doc.createElement("span");
                  span.setAttribute("style", "font-family: monospace;");
                  span.appendChild(doc.createTextNode
                                   (graphicsFiles[filesProcessed]));
                  td.appendChild(span);
                }
                tr.appendChild(td);
                filesProcessed++;
              }
              // we've listed all the files, so create an empty column
              else if (emptyColumnNotCreated)
              {
                td = doc.createElement("td");
                td.setAttribute("colspan",
                                String.valueOf(imagesPerRowValue - i));
                td.setAttribute("bgcolor", "#CCCCCC");
                tr.appendChild(td);
                emptyColumnNotCreated = false;
              }
            }
            table.appendChild(tr);
          }
        }
      }
      return table;
    }
  }
```

Notice that the extension element receives two arguments from Xalan: `org.apache.xalan.extensions.XSLTProcessorContext` and `org.apache.xalan.templates.ElemExtensionCall`. We use the `XSLTProcessorContext` to access the `<photo-album>` element in the XML source document. Xalan and Saxon provide similar context objects to extension elements and functions, although they have different structures, methods, and capabilities.

If you've written Java code to manipulate the Document Object Model (DOM), you'll recognize the techniques at work here. We create a DOM element named `table`, then a DOM element named `tr`, then some DOM elements named `td`, then we put text or `img` elements inside them, and so forth. It's tedious, as DOM code typically is, but it's straightforward.

When we create a `td` element, for example, we create an `a` element, and then we set its attributes. Next, we create an `img` element and set the `img` element's attributes. When the `img` element is complete, we add it to the `a` element, then we add the `a` element to the `td` element. If the XML source specified that we should include the filenames in the

HTML table, we create a text node that contains the filename and add it to the `td` element. When the `td` element is complete, we add it to the `tr` element; when the `tr` is complete, we add it to the `table` element. At the end of the code, the `table` element is complete, so we return that element to the XSLT processor. The processor then uses that element and all of its children in the output document.

To simplify the Java code, we use a Java `FilenameFilter` that selects only the files we want. The extension element invoked this class with this line of Java code:

```
String[] graphicsFiles = dir.list(new GraphicsFilenameFilter());
```

If it's not a graphical file, we don't want to put it into the album. Here's the code for the filter:

```
/*
 * GraphicsFilenameFilter.java
 * Created on Nov 28, 2006 by Doug Tidwell
 */

package com.oreilly.xslt.xalan;
// The filename filter for Saxon is identical except for the package
// name (com.oreilly.xslt.saxon)

import java.io.File;
import java.io.FilenameFilter;

public class GraphicsFilenameFilter implements FilenameFilter
{

  public boolean accept(File dir, String name)
  {
    String lcName = name.toLowerCase();
    if (lcName.endsWith(".jpg")  ||
        lcName.endsWith(".jpeg") ||
        lcName.endsWith(".png")  ||
        lcName.endsWith(".gif")  ||
        lcName.endsWith(".bmp"))
      return true;
    else
      return false;
  }

}
```

If the filename extension is `.jpg`, `.jpeg`, `.png`, `.gif` or `.bmp`, it will pass the filter. Any file that doesn't pass the filter doesn't appear in our list of filenames. We use the filename filter when we create the list of files we have to process:

```
String[] graphicsFiles = dir.list(new GraphicsFilenameFilter());
```

We're assuming that any file with the appropriate extension is actually an image. More ambitious readers are welcome to add code that actually opens each file to verify that it is an image. Another useful feature would be to check the dimensions of the images and create a thumbnail that preserves the aspect ratio. Finally, JPEG files can contain metadata about the date and time that the picture was taken; displaying that information in the HTML output would be a nice touch as well.

Saxon Java Version

The Saxon extension element is considerably more complicated than the Xalan version. We have to create a class that implements the `net.sf.saxon.style.ExtensionElementFactory` interface. That class is responsible for processing the attributes of the `<SaxonPhotoAlbum>` element, including processing any attribute value templates that are in the element. Here's the short file that performs this step:

```
/*
 * PhotoAlbumElementFactory.java
 * Created on Nov 28, 2006 by Doug Tidwell
 */

package com.oreilly.xslt.saxon;

import net.sf.saxon.style.ExtensionElementFactory;

public class PhotoAlbumElementFactory
  implements ExtensionElementFactory
{
  public Class getExtensionClass(String elementName)
  {
    if (elementName.equals("SaxonPhotoAlbum"))
      return SaxonPhotoAlbum.class;
    return null;
  }
}
```

Notice that the code returns the `class` object for the `SaxonPhotoAlbum` class. Saxon uses that object to actually create a `SaxonPhotoAlbum` object that processes the element in the stylesheet:

```
/*
 * SaxonPhotoAlbum.java
 * Created on Nov 28, 2006 by Doug Tidwell
 */

package com.oreilly.xslt.saxon;

import java.io.File;

import net.sf.saxon.event.Receiver;
import net.sf.saxon.event.ReceiverOptions;
import net.sf.saxon.expr.Expression;
```

```
import net.sf.saxon.expr.SimpleExpression;
import net.sf.saxon.expr.XPathContext;
import net.sf.saxon.instruct.Executable;
import net.sf.saxon.om.NamePool;
import net.sf.saxon.style.ExtensionInstruction;
import net.sf.saxon.style.StandardNames;
import net.sf.saxon.trans.XPathException;

public class SaxonPhotoAlbum
  extends ExtensionInstruction
{
  Expression directory;
  Expression imagesPerRow;
  Expression includeFilenames;

  @Override
  public void prepareAttributes() throws XPathException
  {
    String directoryAttr = attributeList.getValue("", "directory");
    if (directoryAttr == null)
    {
      compileError("The directory attribute is required.", "");
      directoryAttr = "";
    }
    directory = makeAttributeValueTemplate(directoryAttr);

    String imagesPerRowAttr =
      attributeList.getValue("", "imagesPerRow");
    if (imagesPerRowAttr == null)
      imagesPerRowAttr = "5";
    imagesPerRow = makeAttributeValueTemplate(imagesPerRowAttr);

    String includeFilenamesAttr =
      attributeList.getValue("", "includeFilenames");
    if (includeFilenamesAttr == null)
      includeFilenamesAttr = "no";
    includeFilenames = makeAttributeValueTemplate(includeFilenamesAttr);
  }

  @Override
  public Expression compile(Executable arg0) throws XPathException
  {
    SaxonPhotoAlbumInstruction spa =
      new SaxonPhotoAlbumInstruction(directory,
                                     imagesPerRow,
                                     includeFilenames);
    return spa;
  }

  private static class SaxonPhotoAlbumInstruction
    extends SimpleExpression {

    private static final long serialVersionUID = 1184239671886575198L;
    public static final int DIRECTORY = 0;
    public static final int IMAGES_PER_ROW = 1;
```

```
public static final int INCLUDE_FILENAMES = 2;

public SaxonPhotoAlbumInstruction(Expression directory,
    Expression imagesPerRow, Expression includeFilenames)
{
  Expression[] subs = {directory, imagesPerRow, includeFilenames};
  setArguments(subs);
}

/**
 * A subclass must provide one of the methods evaluateItem(),
 * iterate(), or process(). This method indicates which of
 * the three is provided.
 */

public int getImplementationMethod()
{
    return Expression.PROCESS_METHOD;
}

public String getExpressionType()
{
    return "spa:SaxonPhotoAlbum";
}

public void process(XPathContext context) throws XPathException
{
  String directoryName =
    arguments[DIRECTORY].evaluateAsString(context);

  int imagesPerRowValue = 5;
  try
  {
    imagesPerRowValue =
        Integer.parseInt(arguments[IMAGES_PER_ROW].
                         evaluateAsString(context));
  }
  catch (NumberFormatException nfe)
  {
    imagesPerRowValue = 5;
  }
  if (!(imagesPerRowValue > 0))
  {
    imagesPerRowValue = 5;
  }

  boolean includeFilenamesFlag =
    arguments[INCLUDE_FILENAMES].evaluateAsString(context)
    .equals("yes");

  NamePool pool = context.getController().getNamePool();
  int anchorCode = pool.allocate("", "", "a");
  int borderCode = pool.allocate("", "", "border");
  int brCode = pool.allocate("", "", "br");
  int cellpaddingCode = pool.allocate("", "", "cellpadding");
```

```
int colspanCode = pool.allocate("", "", "colspan");
int heightCode = pool.allocate("", "", "height");
int hrefCode = pool.allocate("", "", "href");
int imgCode = pool.allocate("", "", "img");
int spanCode = pool.allocate("", "", "span");
int srcCode = pool.allocate("", "", "src");
int styleCode = pool.allocate("", "", "style");
int tableCode = pool.allocate("", "", "table");
int colCode = pool.allocate("", "", "td");
int rowCode = pool.allocate("", "", "tr");
int widthCode = pool.allocate("", "", "width");

Receiver out = context.getReceiver();

// start <table>
out.startElement(tableCode, StandardNames.XDT_UNTYPED, locationId, 0);
out.attribute(borderCode, StandardNames.XDT_UNTYPED,
              "3", locationId, 0);
out.attribute(cellpaddingCode, StandardNames.XDT_UNTYPED,
              "5", locationId, 0);

String[] graphicsFiles;
int numFiles = 0;

File dir = new File(directoryName);

// Return an error message if the directory doesn't exist
if (!dir.exists())
{
  // start <tr>
  out.startElement(rowCode, StandardNames.XDT_UNTYPED, locationId, 0);

  // start <td>
  out.startElement(colCode, StandardNames.XDT_UNTYPED, locationId, 0);
  out.attribute(styleCode, StandardNames.XDT_UNTYPED,
                "font-weight: bold; font-size: 150%;", locationId, 0);
  out.characters("The directory ", locationId, 0);

  // start <span>
  out.startElement(spanCode, StandardNames.XDT_UNTYPED, locationId, 0);
  out.attribute(styleCode, StandardNames.XDT_UNTYPED,
                "font-family: monospace;", locationId, 0);
  out.characters(directoryName, locationId, 0);
  out.endElement(); // end <span>

  out.characters(" does not exist!", locationId, 0);
  out.endElement(); // end <td>

  out.endElement(); // end <tr>
}
else
{
  // Use the filename filter to find the images
  graphicsFiles = dir.list(new GraphicsFilenameFilter());
  if (graphicsFiles != null)
```

```
  numFiles = graphicsFiles.length;

// Return an error message if the directory doesn't contain
// any images
if (numFiles == 0)
{
  // start <tr>
  out.startElement(rowCode, StandardNames.XDT_UNTYPED, locationId, 0);
  // start <td>
  out.startElement(colCode, StandardNames.XDT_UNTYPED, locationId, 0);
  out.attribute(styleCode, StandardNames.XDT_UNTYPED,
                "font-weight: bold; background: #CCCCCC; " +
                "font-size: 150%;",
                locationId, 0);
  out.characters("The directory ", locationId, 0);
  // start <span>
  out.startElement(spanCode, StandardNames.XDT_UNTYPED, locationId, 0);
  out.attribute(styleCode, StandardNames.XDT_UNTYPED,
                "font-family: monospace;", locationId, 0);
  out.characters(directoryName, locationId, 0);
  out.endElement(); // end <span>
  out.characters(" doesn't contain any images!", locationId, 0);

  out.endElement(); // end <td>

  out.endElement(); // end <tr>
}

// We've got images, so let's process 'em...
else
{
  // start <tr>
  out.startElement(rowCode, StandardNames.XDT_UNTYPED, locationId, 0);

  // start <td>
  out.startElement(colCode, StandardNames.XDT_UNTYPED, locationId, 0);
  out.attribute(colspanCode, StandardNames.XDT_UNTYPED, String
      .valueOf(imagesPerRowValue), locationId, 0);
  out.attribute(styleCode, StandardNames.XDT_UNTYPED,
                "font-weight: bold; background: #CCCCCC; "
                    + "font-size: 150%;", locationId, 0);
  out.characters("Photos from ", locationId, 0);

  // start <span>
  out.startElement(spanCode, StandardNames.XDT_UNTYPED, locationId, 0);
  out.attribute(styleCode, StandardNames.XDT_UNTYPED,
                "font-family: monospace;", locationId, 0);
  out.characters(directoryName + ":", locationId, 0);
  out.endElement(); // end <span>

  out.endElement(); // end <td>

  out.endElement(); // end <tr>
}
```

```
int filesProcessed = 0;
boolean emptyColumnNotCreated = true;

while (filesProcessed < numFiles)
{
  // start <tr>
  out.startElement(rowCode, StandardNames.XDT_UNTYPED, locationId, 0);

  for (int i = 0; i < imagesPerRowValue; i++)
  {
    if (filesProcessed < numFiles)
    {
      // start <td>
      out.startElement(colCode, StandardNames.XDT_UNTYPED, locationId,
                       0);
      out.attribute(styleCode, StandardNames.XDT_UNTYPED,
                    "text-align: center;", locationId, 0);

      String qualifiedFilename = "file:\\\\" + directoryName
          + File.separatorChar + graphicsFiles[filesProcessed];

      // start <a>
      out.startElement(anchorCode, StandardNames.XDT_UNTYPED,
                       locationId, 0);
      out.attribute(hrefCode, StandardNames.XDT_UNTYPED,
                    qualifiedFilename, locationId, 0);

      // start <img>
      out.startElement(imgCode, StandardNames.XDT_UNTYPED, locationId,
                       0);
      out.attribute(widthCode, StandardNames.XDT_UNTYPED, "100px",
                    locationId, 0);
      out.attribute(heightCode, StandardNames.XDT_UNTYPED, "130px",
                    locationId, 0);
      out.attribute(srcCode, StandardNames.XDT_UNTYPED,
                    qualifiedFilename, locationId, 0);
      out.attribute(borderCode, StandardNames.XDT_UNTYPED, "0",
                    locationId, 0);
      out.endElement(); // end <img>

      out.endElement(); // end <a>

      if (includeFilenamesFlag)
      {
        // start <br>
        out.startElement(brCode, StandardNames.XDT_UNTYPED, locationId,
                         0);
        out.endElement(); // end <br>

        // start <span style="font-family: monospace">
        out.startElement(spanCode, StandardNames.XDT_UNTYPED,
                         locationId, 0);
        out.attribute(styleCode, StandardNames.XDT_UNTYPED,
                      "font-family: monospace;", locationId, 0);
        out.characters(graphicsFiles[filesProcessed], locationId, 0);
```

```
                out.endElement(); // end <span>
              }
              out.endElement(); // end <td>
              filesProcessed++;
            }
            // We've listed all the files, so create an empty column
            else if (emptyColumnNotCreated)
            {
              // start <td>
              out.startElement(colCode, StandardNames.XDT_UNTYPED, locationId,
                               0);
              out.attribute(styleCode, StandardNames.XDT_UNTYPED,
                            "background: #CCCCCC;", locationId, 0);
              out.attribute(colspanCode, StandardNames.XDT_UNTYPED,
                            String.valueOf(imagesPerRowValue - i),
                            locationId, 0);
              out.characters(" ", locationId,
                             ReceiverOptions.DISABLE_ESCAPING);
              out.endElement(); // end <td>
              emptyColumnNotCreated = false;
            }
          }
          out.endElement(); // end <tr>
        }
      }
      out.endElement(); // end <table>
    }
  }
}
```

As with the Xalan extension, Saxon provides us with a context object—in this case, a Saxon `net.sf.saxon.expr.XPathContext`. The Saxon extension works similarly to the Xalan extension, creating elements and attributes and returning them as a tree structure. There are differences, though; to start, we need to define a Saxon `NamePool`:

```
NamePool pool = context.getController().getNamePool();
int anchorCode = pool.allocate("", "", "a");
int borderCode = pool.allocate("", "", "border");
int brCode = pool.allocate("", "", "br");
int cellpaddingCode = pool.allocate("", "", "cellpadding");
...
```

We're defining the names of the elements and attributes we're going to create. Next, we use the `out` object to create the various elements of the table. The `out` object is a Saxon `net.sf.saxon.event.Receiver` object that we retrieve from the `XPathContext`. We use `out` to create the elements, attributes, and text nodes. The four methods we use from the `Receiver` class are `startElement( )`, `attribute( )`, `characters( )`, and `endElement( )`. Notice that the `endElement( )` method doesn't take any parameters; it simply ends whatever element is being created. The code is commented to indicate whenever an element is started or ended.

As we mentioned with the Xalan version of this extension, we use a `FilenameFilter` to select only the image files.

.NET Version

The .NET version of the photo album is slightly different. We'll implement the extension as an extension *function*, not as an extension element. We'll do this to show an alternate technique for creating nodes in the output tree.

Our extension function creates a DOM tree that contains all of the HTML elements needed for the output tree. The Xalan and Saxon versions of this extension `element` replace the extension element in the stylesheet with the appropriate generated nodes. In the .NET environment, our extension function returns a DOM tree that is added to the result tree.

Here's the code for the extension function. It follows the same basic procedure as the other versions of the extension: use system functions to get the list of files and then create the DOM elements for the HTML table:

```
/*
 * PhotoAlbum.cs
 * Created on Nov 28, 2006 by Doug Tidwell
 */

using System;
using System.IO;
using System.Text;
using System.Xml;

// This class creates a photo album with all of the images in a
// given directory.
namespace com.oreilly.xslt
{
  class PhotoAlbum
  {
    // getPhotoListing is the name of the extension function.  Notice
    // that it returns an XMLNode; that node is inserted into the
    // output tree.
    public XmlNode getPhotoListing(String dirName,
                                   String imgPerRow,
                                   String incFilenames)
    {
      // For parameters, we get the name of the directory, the number
      // of images per row, and whether the names of the files should
      // be displayed.
      int imagesPerRow = int.Parse(imgPerRow);
      Boolean includeFilenames = (incFilenames.Equals("yes"));

      // We're building a DOM tree, effectively, so we create some
      // elements beforehand.
      XmlElement anchorEl, breaklineEl, imageEl, spanEl,
          tableEl, tableDataEl, tableRowEl;
      XmlNode textNode;

      XmlDocument doc = new XmlDocument();
      XmlDocumentFragment df = doc.CreateDocumentFragment();
      tableEl = doc.CreateElement("table");
```

```
tableEl.SetAttribute("border", "3");
tableEl.SetAttribute("cellpadding", "5");

Boolean directoryExists = true;

System.Collections.ArrayList allGraphicsFiles =
    new System.Collections.ArrayList();

// We add all of the JPEG, PNG, GIF and BMP files in the
// specified directory.  Windows isn't case-sensitive, so
// the code is slightly simpler than the Java version.
try
{
  allGraphicsFiles.InsertRange(allGraphicsFiles.Count,
      System.IO.Directory.GetFiles(dirName, "*.jpg"));
  allGraphicsFiles.InsertRange(allGraphicsFiles.Count,
      System.IO.Directory.GetFiles(dirName, "*.jpeg"));
  allGraphicsFiles.InsertRange(allGraphicsFiles.Count,
      System.IO.Directory.GetFiles(dirName, "*.png"));
  allGraphicsFiles.InsertRange(allGraphicsFiles.Count,
      System.IO.Directory.GetFiles(dirName, "*.gif"));
  allGraphicsFiles.InsertRange(allGraphicsFiles.Count,
      System.IO.Directory.GetFiles(dirName, "*.bmp"));
}

// Return an error message if the directory doesn't exist
catch (System.IO.DirectoryNotFoundException dnfe)
{
  directoryExists = false;
  tableRowEl = doc.CreateElement("tr");
  tableDataEl = doc.CreateElement("td");
  tableDataEl.SetAttribute("style", "font-weight: bold; " +
      "font-size: 150%;");
  textNode = doc.CreateTextNode("The directory ");
  tableDataEl.AppendChild(textNode);
  spanEl = doc.CreateElement("span");
  spanEl.SetAttribute("style", "font-family: monospace;");
  spanEl.AppendChild(doc.CreateTextNode(dirName));
  tableDataEl.AppendChild(spanEl);
  textNode = doc.CreateTextNode(" doesn't exist!");
  tableDataEl.AppendChild(textNode);
  tableRowEl.AppendChild(tableDataEl);
  tableEl.AppendChild(tableRowEl);
  df.AppendChild(tableEl);
}

int numFiles = allGraphicsFiles.Count;

if (directoryExists)
{
  // Return an error message if the directory doesn't contain
  // any images
  if (numFiles == 0)
  {
    tableRowEl = doc.CreateElement("tr");
```

```
  tableDataEl = doc.CreateElement("td");
  tableDataEl.SetAttribute("style", "font-weight: bold; " +
      "font-size: 150%;");
  textNode = doc.CreateTextNode("The directory ");
  tableDataEl.AppendChild(textNode);
  spanEl = doc.CreateElement("span");
  spanEl.SetAttribute("style", "font-family: monospace;");
  spanEl.AppendChild(doc.CreateTextNode(dirName));
  tableDataEl.AppendChild(spanEl);
  textNode =
      doc.CreateTextNode(" doesn't contain any images!");
  tableDataEl.AppendChild(textNode);
  tableRowEl.AppendChild(tableDataEl);
  tableEl.AppendChild(tableRowEl);
  df.AppendChild(tableEl);
}

// We've got some images, so let's process 'em...
else
{
  // First we create a table row for the header.
  tableRowEl = doc.CreateElement("tr");
  tableDataEl = doc.CreateElement("td");
  tableDataEl.SetAttribute("colspan", imgPerRow);
  tableDataEl.SetAttribute("style", "font-weight: bold; " +
      "background: #CCCCCC; font-size: 150%;");
  tableDataEl.AppendChild(doc.CreateTextNode("Photos from "));
  spanEl = doc.CreateElement("span");
  spanEl.SetAttribute("style", "font-family: monospace;");
  spanEl.AppendChild(doc.CreateTextNode(dirName + ":"));
  tableDataEl.AppendChild(spanEl);
  tableRowEl.AppendChild(tableDataEl);
  tableEl.AppendChild(tableRowEl);

  // Now we process all of the files.  The emptyColumnNotCreated
  // flag tells us whether the blank cell has been created.
  // If there are five images per row and there are 47 images,
  // the last row will contain a blank cell with a colspan of 3.
  // We only create that blank cell once.
  int filesProcessed = 0;
  Boolean emptyColumnNotCreated = true;

  // We look through all of the files
  while (filesProcessed < numFiles)
  {
    tableRowEl = doc.CreateElement("tr");

    // Process the next {imagesPerRow} images
    for (int i = 0; i < imagesPerRow; i++)
    {
      if (filesProcessed < numFiles)
      {
        tableDataEl = doc.CreateElement("td");
        tableDataEl.SetAttribute("style", "text-align: center;");
        String qualifiedFilename = "file:\\\\" +
```

```
                    allGraphicsFiles[filesProcessed];
                anchorEl = doc.CreateElement("a");
                imageEl = doc.CreateElement("img");
                imageEl.SetAttribute("src", qualifiedFilename);
                imageEl.SetAttribute("width", "100px");
                imageEl.SetAttribute("height", "130px");
                imageEl.SetAttribute("border", "0");
                anchorEl.AppendChild(imageEl);
                tableDataEl.AppendChild(anchorEl);

                if (includeFilenames)
                {
                  breaklineEl = doc.CreateElement("br");
                  tableDataEl.AppendChild(breaklineEl);

                  // We use FileInfo to get the filename without the path.
                  FileInfo fi =
                    new FileInfo((String)allGraphicsFiles[filesProcessed]);
                  spanEl = doc.CreateElement("span");
                  spanEl.SetAttribute("style", "font-family: monospace;");
                  spanEl.AppendChild(doc.CreateTextNode(fi.Name));
                  tableDataEl.AppendChild(spanEl);
                }
                tableRowEl.AppendChild(tableDataEl);
                filesProcessed++;
              }

              // If we've processed all of the files and we haven't
              // created the empty column, we'll do that now.
              else if (emptyColumnNotCreated)
              {
                tableDataEl = doc.CreateElement("td");
                Int32 colspan = (imagesPerRow - i);
                tableDataEl.SetAttribute("colspan", colspan.ToString());
                tableDataEl.SetAttribute("style", "background: #CCCCCC;");
                tableRowEl.AppendChild(tableDataEl);
                emptyColumnNotCreated = false;
              }
            }

            // Throughout this code, we create elements and append them to
            // the appropriate parent element.
            tableEl.AppendChild(tableRowEl);
          }
          df.AppendChild(tableEl);
        }
      }

      // Once we've finished, we have a node-set that contains all of the
      // HTML elements we need.  That is returned to the caller, which
      // then inserts those nodes into the output stream.
      return df;
    }
  }
}
```

The DOM code is more similar to the Xalan extension than to the Saxon one. We're using .NET classes, but the overall flow of the code is similar to the two Java extensions. We collect the filenames, create a table, create a cell for each graphic, and then return the table element. In this case, we return a .NET `XmlDocumentFragment`.

Now that we have our extension function defined, we need a stylesheet that invokes it. As we've seen before, we need to define a namespace for the extension:

```
<?xml version="1.0" encoding="utf-8" ?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:ora="http://www.oreilly.com/xslt"
  extension-element-prefixes="ora">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Photo album extension function test</title>
      </head>
      <body style="font-family: sans-serif;">
        <h1>
          <xsl:text>.Net Photo album extension function test</xsl:text>
        </h1>
        <xsl:variable name="album"
                      select="ora:getPhotoListing(/photo-album/@directory,
                      /photo-album/@imagesPerRow,
                      /photo-album/@includeFilenames)"/>
        <xsl:copy-of select="$album"/>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

Because our .NET implementation is an extension function, we assign its value to a variable. Once the variable (a result tree fragment) is created, we use `<xsl:copy-of>` to copy everything in the variable to the output tree.

As with our earlier .NET examples, we need a C# file that creates an `XslCompiledTransformation` object. We also create a `PhotoAlbum` object and associate it with the namespace prefix defined in our stylesheet. The final step is to create an `XsltArgumentList` that ties everything together. Here's the code:

```
/*
 * Main.cs
 * Created Nov 22, 2006 by Doug Tidwell
 */

using System;
using System.Text;
using System.Xml;
using System.Xml.Xsl;
```

```
namespace com.oreilly.xslt
{
  class MainClass
  {
    static void Main(string[] args)
    {
      // Create the stylesheet object and the XMLWriter that
      // writes the output to a file
      XslCompiledTransform stylesheet = new XslCompiledTransform();
      XmlTextWriter xWriter =
          new XmlTextWriter(args[2], Encoding.UTF8);

      // Use an XsltSettings object that allows executing scripts
      // (we need this for extensions), then load the stylesheet
      XsltSettings settings = new XsltSettings(true, true);
      stylesheet.Load(args[1], settings, new XmlUrlResolver());

      // Create an extension object
      PhotoAlbum pa = new PhotoAlbum();

      // We add the PhotoAlbum object as an extension object to the
      // XsltArgumentList.
      XsltArgumentList argList = new XsltArgumentList();
      argList.AddExtensionObject("http://www.oreilly.com/xslt", pa);

      // With everything in place, we call the Transform() method
      // to do the work...
      stylesheet.Transform(args[0], argList, xWriter);
    }
  }
}
```

Notice that the namespace defined in the stylesheet, `http://www.oreilly.com/xslt`, is the same namespace we use in the `XsltArgumentList` we create in the C# code. If these two don't match, the .NET XSLT processor won't be able to find the extension class.

When everything works together, the results appear as in Figure 9-14.

Most digital cameras produce JPEG files that contain EXIF (Exchangeable Image Format) tags. Those tags contain information such as the model of camera used to take the picture, the date and time of the picture, and other aspects. If you'd like to delve into all the details of an EXIF-tagged JPEG file, see the EXIF Tag Parsing Library here: *http://sourceforge.net/projects/libexif*. Enhancing the extension so that those details appear in the table (or are displayed in a small window when you move the mouse over the picture) would be a nice addition to the code.

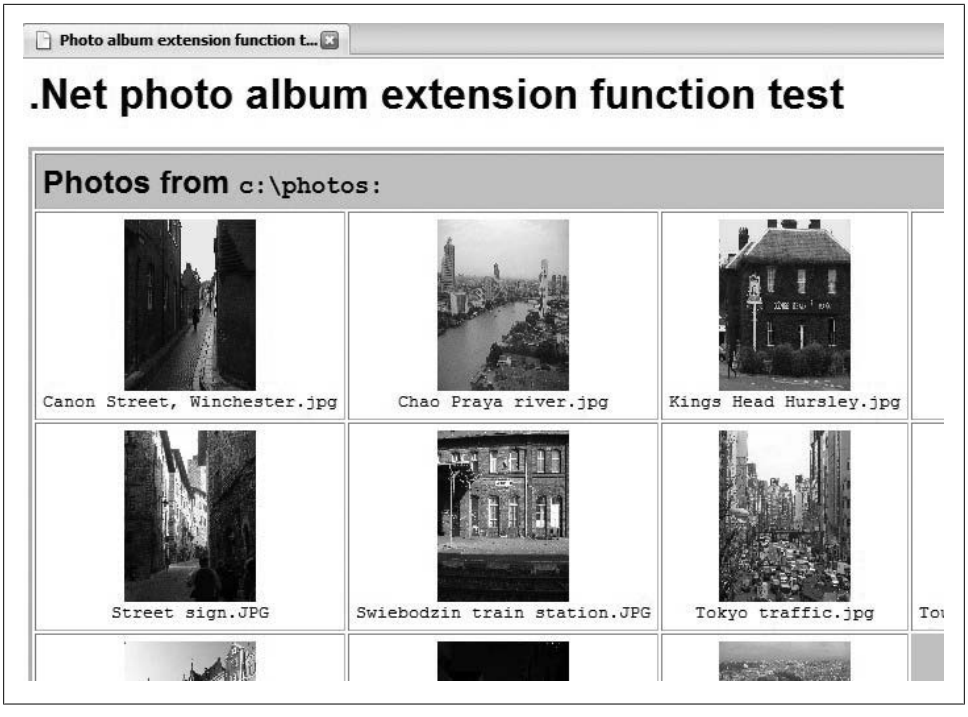


Figure 9-14. A photo album generated by a .NET extension function

Summary

In this chapter, we've run the gamut of extension functions and extension elements, demonstrating how to add sophisticated processing power to our stylesheets. We've generated many output files from a stylesheet, created JPEGs dynamically, created interactive graphics, interacted with databases, and generated HTML files created with data retrieved from the filesytem.

In terms of XSLT itself, we've created both extension elements and extension functions, and we've returned simple values as well as trees of DOM objects. Most importantly, we've demonstrated how to access the XML source document and the extension code in a variety of ways, using different platforms, languages, and XSLT processors. If you master these techniques, there's practically nothing you can't do in your stylesheets.

APPENDIX A
XSLT Reference

This chapter is a complete reference to all the elements defined in the XSLT specification. To keep the examples simple, most of them generate plain text instead of XML, HTML, or XHTML. That allows us to focus on the XSLT itself instead generating CSS properties, HTML tables, or other details. Where an XML, HTML, or XHTML example is more instructive, we'll use it of course, but most stylesheets are as simple as possible.

[2.0] Attributes common to all XSLT elements

There are six standard attributes that can be used on any XSLT element; we'll define them here rather than redefine them for every XSLT element. While these attributes can be used on any XSLT element, they are normally used on elements such as `<xsl:stylesheet>` and `<xsl:template>`.

Attributes

`version`
: Defines the version of XSLT used to process this element. This is useful when you want a particular XSLT element to be processed using the rules of a particular version of the standard. For example, `<xsl:value-of select="1 div 0"/>` works differently in XSLT 1.0 and 2.0. XSLT 2.0 treats this as a fatal error, while XSLT 1.0 returns `Infinity`. Using `<xsl:value-of version="1.0" ...>` ensures that version 1.0 processing is used.

Although the `<xsl:output>` element has a `version` attribute, it specifies the value of the `version` attribute in the output. For example, `<xsl:output method="xml" version="1.1">` creates a result document with an XML declaration of `<?xml version="1.1" ...?>`.

`exclude-result-prefixes`
: Lists the prefixes of namespaces that should not be copied to the output. Typically used on the `<xsl:stylesheet>` element only.

`extension-element-prefixes`
: Defines the namespace prefixes that identify extension elements. The XSLT processor uses these namespaces to identify code that provides additional processing. Typically used on the `<xsl:stylesheet>` element only.

`xpath-default-namespace`
: Defines the default namespace used in XPath expressions and patterns. If you're transforming a document that uses a default namespace (`http://www.oreilly.com`, for example), you must namespace-qualify the names of any elements in that document. Defining `xpath-default-namespace="http://www.oreilly.com"` tells XPath to use this namespace as the default for element and type names in XPath expressions. If the documents you're transforming use different default namespaces in different parts of the document, you can use this attribute to change the default XPath namespace. This is typically used on the `<xsl:stylesheet>` element only.

`default-collation`
: A series of space-separated URIs that define the default collation sequence. The default collation sequence is used by `<xsl:key>` and `<xsl:for-each-group>`, but it does not affect the collation used by `<xsl:sort>`. The way collation sequences are defined varies from one XSLT processor to another, so check your processor's documentation for information on defining a collation sequence.

`use-when`
: Defines a statement involving a `system-property` that must be true for this XSLT element to be processed. As an example, using the function `system-property('xsl:is-schema-aware')` lets you determine whether the XSLT processor is schema-aware.

[2.0] <xsl:analyze-string>

Allows you to compare a string and a regular expression.

Changes in XSLT 2.0

`<xsl:analyze-string>` is new to XSLT 2.0.

Category

Instruction.

Required Attributes

`select`
: An XPath expression that defines the string to be analyzed. The expression is converted to a string if necessary.

`regex`
: The regular expression. Regular expressions commonly use curly braces ({ and }), which XSLT interprets as the start and end of an attribute value template. For this reason, any curly braces in a regular expression must be doubled. For example, the regular expression `"[0-9]{{5}}"` matches a five-digit number, while `"[0-9]{5}"` matches a one-digit number followed by the number `5`.

It is a fatal error if the regular expression matches a zero-length string. See Appendix E for more details.

Optional Attribute

`flags`
: The `flags` attribute modifies how the regular expression is processed. There are four different flags:

  `s`
  : Regular expressions are evaluated in what the specs refer to as "dot-all" mode. When this flag is used, the dot operator (`.`) matches any character. Under normal processing (without the `s` flag), the dot operator matches any character *except* the newline character (`#xA`). This flag is useful when you want to match strings that might include a newline character.

  `m`
  : Regular expressions are evaulated in multiline mode. By default, the meta-character (^) matches the start of the entire string, while $ matches the end of the entire string. In multiline mode, ^ matches the start of any line within the string and $ matches the end of any line within the string.

  `i`
  : Regular expressions are evaluated in case-insensitive mode. The regular expression `"a"` matches both `"a"` and `"A"`.

    Note that Unicode issues can complicate this greatly. For example, the XQuery 1.0 and XPath 2.0 Functions and Operators spec gives the example of the Unicode sign for degrees Kelvin (`K`), which is the letter `"K"`. The combination of `regex="k"` and `flags="i"` matches the Kelvin sign as well as the letters `"k"` (`k`) and `"K"` (`K`).

    Other Unicode characters don't convert to letters. For example, the Unicode symbol for the Roman numeral `I` (`Ⅰ`) looks like the letter `I`, but does not convert to that. Fortunately, these complications are beyond the scope of this book.

x

All whitespace characters (#x9, #xA, #xD, and #x20) are removed from the regular expression before any comparison is done. In other words, with the x flag, the regular expressions "John Smith" and "JohnSmith" are the same. This flag is useful when you want to break a long regular expression into multiple lines to make it easier to read.

The flags can be combined in any order. The attributes flags="xis" and flags="six" work exactly the same way.

Content

<xsl:analyze-string> can contain a single, optional <xsl:matching-substring> element and/or a single, optional <xsl:non-matching-substring> element. If both <xsl:matching-substring> and <xsl:non-matching-substring> are included, the <xsl:matching-substring> element must appear first. In other words, all three of these <xsl:analyze-string> elements are valid:

```
<xsl:analyze-string select="." regex="a">
  <xsl:matching-substring>...</xsl:matching-substring>
</xsl:analyze-string>

<xsl:analyze-string select="." regex="a">
  <xsl:non-matching-substring>...</xsl:non-matching-substring>
</xsl:analyze-string>

<xsl:analyze-string select="." regex="a">
  <xsl:matching-substring>...</xsl:matching-substring>
  <xsl:non-matching-substring>...</xsl:non-matching-substring>
</xsl:analyze-string>
```

The <xsl:analyze-string> element can also contain any number of <xsl:fallback> elements. If your XSLT 1.0 processor supports forwards-compatible mode, <xsl:fallback> lets you define how the processor should respond when the <xsl:analyze-string> element is not supported. You could have a stylesheet structured like this:

```
<xsl:analyze-string...>
  <xsl:matching-substring>...</xsl:matching-substring>
  <xsl:fallback>
    <!-- Instructions for a 1.0 processor -->
  </xsl:fallback>
</xsl:analyze-string>
```

A forwards-compatible XSLT 1.0 processor will see the <xsl:analyze-string> element, which it cannot process. Whenever the processor finds an unsupported element in the xsl: namespace, it looks inside that element for an <xsl:fallback> element. If it finds one, the processor attempts to process whatever is inside <xsl:fallback>.

Appears in

<xsl:analyze-string> appears inside a template.

Defined in

XSLT section 15, "Regular Expressions."

Regular expressions in XSLT 2.0, XPath 2.0, and XQuery 1.0 are based on the syntax defined in the XML Schema standard; see Appendix F, "Regular Expressions," in *XML Schema Part 2: Datatypes* for the complete syntax. The XQuery 1.0 and XPath 2.0 Functions and Operators spec defines extensions to the XML Schema regular expression syntax in section 7.6, "String Functions that Use Pattern Matching."

Example

Here is an example of `<xsl:analyze-string>` that uses a regular expression to convert U.S. and Canadian telephone numbers of the form `999-999-9999` to the form `+1 (999) 999-9999`:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- analyze-string1.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:for-each select="phonelist/phonenumber">
      <xsl:analyze-string select="."
        regex="([0-9]{{3}})-(\p{{Nd}}{{3}})-([0-9]{{4}})">
        <xsl:matching-substring>
          <xsl:text>&#xA;+1 (</xsl:text>
          <xsl:value-of select="regex-group(1)"/>
          <xsl:text>) </xsl:text>
          <xsl:value-of select="regex-group(2)"/>
          <xsl:text>-</xsl:text>
          <xsl:value-of select="regex-group(3)"/>
        </xsl:matching-substring>
        <xsl:non-matching-substring>
          <xsl:text>&#xA;   Unrecognized phone number: </xsl:text>
          <xsl:value-of select="."/>
        </xsl:non-matching-substring>
      </xsl:analyze-string>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Our stylesheet uses the regular expression `([0-9]{{3}})-(\p{{Nd}}{{3}})-([0-9]{{4}})`. This creates three `regex-group`s; each group is enclosed in a set of parentheses. The code `\p{{Nd}}` uses the named character group `Nd` to refer to numeric digits. See Appendix E for a list of named character groups.

When we use this stylesheet with this list of phone numbers:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- phonelist.xml -->
<phonelist>
  <phonenumber>919-555-1212</phonenumber>
  <phonenumber>(919) 555-1212</phonenumber>
```

```
  <phonenumber>212.555.1212</phonenumber>
  <phonenumber>617-555-1212</phonenumber>
  <phonenumber>+86 555-1212</phonenumber>
</phonelist>
```

we get these results:

```
+1 (919) 555-1212
   Unrecognized phone number: (919) 555-1212
   Unrecognized phone number: 212.555.1212
+1 (617) 555-1212
   Unrecognized phone number: +86 555-1212
```

When our regular expression matches, `regex-group(1)` represents the area code, `regex-group(2)` represents the exchange, and `regex-group(3)` represents the last four digits of the phone number.

If you need to retrieve all of the string that matched the regular expression, `regex-group(0)` does the trick. (Using a dot `"."` does the same thing.) In the previous example, using `regex-group(0)` for the first phone number returns `919-555-1212`.

In some circles, it's fashionable to use periods instead of dashes to separate the sections of a phone number. To handle this, we could change our regular expression to match both periods and dashes:

```
<xsl:analyze-string select="."
  regex="([0-9]{{3}})(-|\.)([0-9]{{3}})(-|\.)([0-9]{{4}})">
```

We replaced the dashes in the original expression with `(-|.)`. This expression matches the third `<phonenumber>` element in addition to the first and fourth, but the output doesn't look right:

```
+1 (919) --555
   Unrecognized phone number: (919) 555-1212
+1 (212) .-555
+1 (617) --555
   Unrecognized phone number: +86 555-1212
```

The problem is that we've created two more groups in the regular expression. We need to change our stylesheet to use `regex-group(1)`, `regex-group(3)`, and `regex-group(5)`:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- analyze-string2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:for-each select="phonelist/phonenumber">
      <xsl:analyze-string select="."
        regex="([0-9]{{3}})(-|\.)([0-9]{{3}})(-|\.)([0-9]{{4}})">
        <xsl:matching-substring>
          <xsl:text>&#xA;+1 (</xsl:text>
          <xsl:value-of select="regex-group(1)"/>
          <xsl:text>) </xsl:text>
          <xsl:value-of select="regex-group(3)"/>
```

```
          <xsl:text>-</xsl:text>
          <xsl:value-of select="regex-group(5)"/>
        </xsl:matching-substring>
        <xsl:non-matching-substring>
          <xsl:text>&#xA;   Unrecognized phone number: </xsl:text>
          <xsl:value-of select="."/>
        </xsl:non-matching-substring>
      </xsl:analyze-string>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Now our updated stylesheet gives us the correct results:

```
+1 (919) 555-1212
   Unrecognized phone number: (919) 555-1212
+1 (212) 555-1212
+1 (617) 555-1212
   Unrecognized phone number: +86 555-1212
```

In our discussion of the `i` flag, we mentioned the example of the Kelvin sign (`#x212A;`) as a special Unicode character that matches the letter `K`. Here's a stylesheet to test that, if you'd like:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- analyze-string3.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:analyze-string select="'&#x212A;'"
      regex="k" flags="i">
      <xsl:matching-substring>
        <xsl:text>It matches!</xsl:text>
      </xsl:matching-substring>
      <xsl:non-matching-substring>
        <xsl:text>It doesn't match!</xsl:text>
      </xsl:non-matching-substring>
    </xsl:analyze-string>
  </xsl:template>

</xsl:stylesheet>
```

For a final example, we'll illustrate what happens with an incorrectly written regular expression. We stated earlier that any curly braces ({ and }) must be doubled. The regular expression `[0-9]{{5}}` matches a five-digit number, while the incorrectly written expression `[0-9]{5}` matches a two-digit number ending in `5`. (The attribute value template `{5}` evaluates to the string `5`.) Here's the stylesheet we'll use:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- analyze-string4.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Test of an incorrectly written regex:&#xA;&#xA;</xsl:text>
    <xsl:call-template name="bad-regex">
      <xsl:with-param name="test-value" select="'37174'"/>
    </xsl:call-template>
    <xsl:call-template name="bad-regex">
      <xsl:with-param name="test-value" select="'95'"/>
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="bad-regex">
    <xsl:param name="test-value" required="yes"/>
    <xsl:value-of select="$test-value"/>
    <xsl:analyze-string select="$test-value" regex="[0-9]{5}">
      <xsl:matching-substring>
        <xsl:text> matches!&#xA;</xsl:text>
      </xsl:matching-substring>
      <xsl:non-matching-substring>
        <xsl:text> doesn't match!&#xA;</xsl:text>
      </xsl:non-matching-substring>
    </xsl:analyze-string>
  </xsl:template>

</xsl:stylesheet>
```

Running this stylesheet against any XML document generates these results:

```
Test of an incorrectly written regex:

37174 doesn't match!
95 matches!
```

The value `95` matches the poorly written expression, whereas the expected `37174` doesn't.

See Also

Appendix E provides complete details on the way regular expressions work in XPath 2.0. Also see the definitions of the following elements and functions: [2.0] `matches( )`, [2.0] `<xsl:matching-substring>`, [2.0] `<xsl:non-matching-substring>`, [2.0] `regex-group( )`, [2.0] `replace( )`, and [2.0] `tokenize( )`.

<xsl:apply-imports>

Allows you to apply to the current node a template imported from another stylesheet.

Category

Instruction.

Required Attributes

None.

Optional Attributes

None.

Content

[1.0] None. `<xsl:apply-imports>` is an empty element.

[2.0] In XSLT 2.0, `<xsl:apply-imports>` can contain zero or more `<xsl:with-param>` elements to pass parameters to an imported template. If you pass extra parameters to an imported template, they are ignored. However, if you don't pass a `required` parameter to an imported template, the XSLT processor throws an error.

Appears in

`<xsl:apply-imports>` appears inside a template.

Defined in

[1.0] XSLT section 5.6, "Overriding Template Rules."

[2.0] XSLT section 6.7, "Overriding Template Rules."

Example

Here is a short XML file we'll use to illustrate `<xsl:apply-imports>`:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- codelisting.xml -->
<chapter>
  <title>Some really great code</title>
  <para>Here is one of my favorite code listings:</para>
  <programlisting>
public class HelloWorld
{
  public static void main(String[] args)
  {
    System.out.println("Hello, World!");
  }
}
  </programlisting>
  <para>I wrote that code all by myself!</para>
</chapter>
```

Our main stylesheet has template rules for the four DocBook elements (`<chapter>`, `<title>`, `<para>`, and `<programlisting>`) in our sample document:

```
<?xml version="1.0"?>
<!-- apply-imports.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:import href="imported.xsl"/>

  <xsl:preserve-space elements="programlisting"/>

  <xsl:output method="html"/>
```

```
  <xsl:template match="chapter">
    <html>
      <head>
        <title>
          <xsl:value-of select="title"/>
        </title>
      </head>
      <body>
        <xsl:apply-templates select="*"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="title">
    <h1>
      <xsl:value-of select="."/>
    </h1>
  </xsl:template>

  <xsl:template match="para">
    <p>
      <xsl:apply-templates select="*|text()"/>
    </p>
  </xsl:template>

  <xsl:template match="programlisting">
    <div style="font-size: 125%; font-weight:bold;
                font-family: monospace;">
      <xsl:apply-imports />
    </div>
  </xsl:template>

</xsl:stylesheet>
```

Here's the stylesheet we'll import:

```
<?xml version="1.0"?>
<!-- imported.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="programlisting">
    <pre>
      <xsl:value-of select="."/>
    </pre>
  </xsl:template>

</xsl:stylesheet>
```

Our imported stylesheet puts `<pre>` tags around any `<programlisting>` it finds. The main stylesheet puts the `<programlisting>` inside a `<div>` that changes the font properties. When we process the XML document with the main stylesheet, we get these results:

```
<!-- codelisting.html -->
<html>
```

Figure A-1. HTML generated using a template imported from another stylesheet

```
    <head>
       <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
       <title>Some really great code</title>
    </head>
    <body>
       <h1>Some really great code</h1>
       <p>Here is one of my favorite code listings:</p>
       <div style="font-size: 125%; font-weight:bold;
                   font-family: monospace;"><pre>
public class HelloWorld
{
  public static void main(String[] args)
  {
    System.out.println("Hello, World!");
  }
}
  </pre></div>
       <p>I wrote that all by myself!</p>
    </body>
</html>
```

The HTML output contains a `<div>` element created by the main stylesheet and a `<pre>` element created by the imported stylesheet (see Figure A-1). If you replace `<xsl:apply-imports />` with `<xsl:value-of select="."/>`, the code listing will be run together on a single line.

See Also

The description of the [2.0] `<xsl:next-match>` element.

<xsl:apply-templates>

Instructs the XSLT processor to apply the appropriate templates to a node-set or sequence.

Category

Instruction.

Required Attributes

None.

Optional Attributes

`select`
: Contains an XPath expression that selects the nodes to which templates should be applied. Valid values include * to select all the *element* children of the current node. Without this attribute, `<xsl:apply-templates>` selects all of the children of the current node, including text, processing instructions, and comments. The instructions `<xsl:apply-templates />` and `<xsl:apply-templates select="node( )"/>` are equivalent.

`mode`
: Defines a *processing mode*, which is a convenient syntax that lets you write specific templates for specific purposes. For example, you could write an `<xsl:template>` with `mode="toc"` to process a node for the table of contents of a document, and write other `<xsl:template>`s with `mode="print"`, `mode="online"`, `mode="index"`, etc. to process the same information for different purposes.

  [2.0] In XSLT 2.0, there are two special values for the `mode` attribute when used with the `<xsl:apply-templates>` element:

  `#default`
  : Matches the default mode

  `#current`
  : Matches the current mode

Content

The `<xsl:apply-templates>` element can contain any number of `<xsl:sort>` and `<xsl:with-param>` elements. In many cases, `<xsl:apply-templates>` is an empty element.

Appears in

`<xsl:apply-templates>` appears inside a template.

Defined in

[1.0] XSLT section 5.4, "Applying Template Rules."

[2.0] XSLT section 6.3, "Applying Template Rules."

Example

Here is a stylesheet that processes the same nodes in three different ways. We invoke each template with a `mode` attribute. We'll use our list of cars as our sample document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- cars.xml -->
<cars>
  <manufacturer name="Chevrolet">
    <car>Cavalier</car>
    <car>Corvette</car>
    <car>Impala</car>
    <car>Malibu</car>
  </manufacturer>
  <manufacturer name="Ford">
    <car>Pinto</car>
    <car>Mustang</car>
    <car>Taurus</car>
  </manufacturer>
  <manufacturer name="Volkswagen">
    <car>Beetle</car>
    <car>Jetta</car>
    <car>Passat</car>
    <car>Touraeg</car>
  </manufacturer>
</cars>
```

Here's our stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apply-templates1.xsl -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Using the mode attribute</title>
      </head>
      <body style="font-family: sans-serif;">
        <table style="text-align: center;" border="1">
          <tr style="font-weight: bold; font-size: 150%;">
            <td width="30%">Default mode</td>
            <td width="30%">Blue mode</td>
            <td width="30%">Red mode</td>
          </tr>
          <tr>
            <td>
              <p>
                <xsl:apply-templates
                  select="/cars/manufacturer"/>
              </p>
            </td>
            <td>
```

```
            <p>
              <xsl:apply-templates mode="blue"
                select="/cars/manufacturer"/>
            </p>
          </td>
          <td>
            <p>
              <xsl:apply-templates mode="red"
                select="/cars/manufacturer"/>
            </p>
          </td>
        </tr>
      </table>
    </body>
  </html>
</xsl:template>

<xsl:template match="manufacturer">
  <div style="color: green; font-style: italic; font-size: 125%;">
    <xsl:apply-templates select="car"/>
  </div>
</xsl:template>

<xsl:template match="manufacturer" mode="blue">
  <div style="color: blue; font-weight: bold;">
    <xsl:apply-templates select="car"/>
  </div>
</xsl:template>

<xsl:template match="manufacturer" mode="red">
  <div style="color: red; font-family: monospace; font-weight: bold;
              font-size: 150%;">
    <xsl:apply-templates select="car"/>
  </div>
</xsl:template>

<xsl:template match="car">
  <xsl:value-of select="."/>
  <br/>
</xsl:template>

</xsl:stylesheet>
```

Our stylesheet generates an HTML document that looks like Figure A-2.

The first template doesn't have a `mode` attribute, so it is the default template.

[2.0] Next we'll take a look at the new features of the `mode` attribute in XSLT 2.0. This stylesheet uses the `#default`, `#current`, and `#all` values of the `mode` attribute:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apply-templates2.xsl -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">

  <xsl:output method="html"/>
```

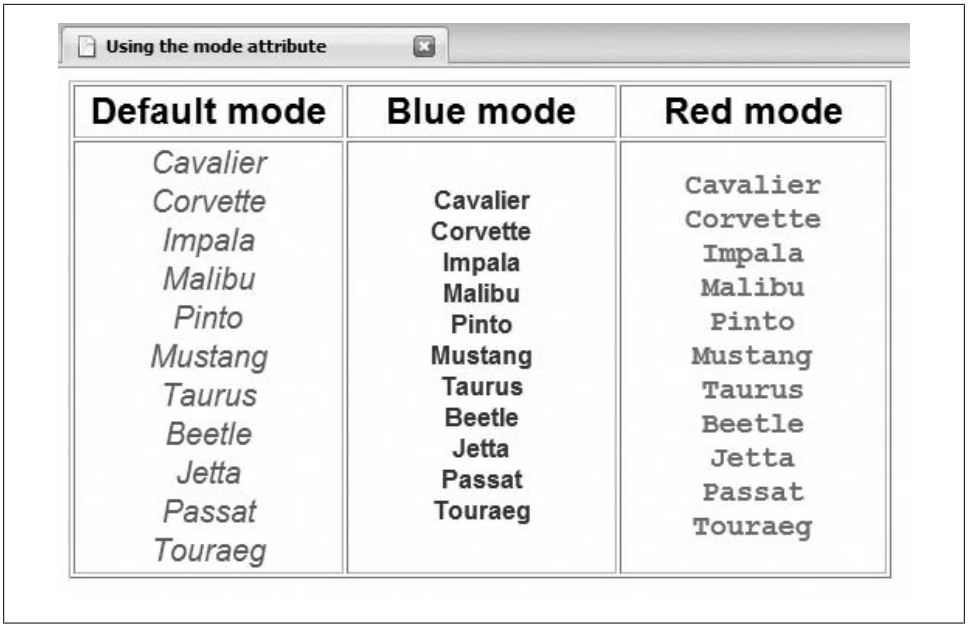


Figure A-2. HTML generated using templates with different modes

```
<xsl:template match="/">
  <html>
    <head>
      <title>Using the mode attribute</title>
    </head>
    <body style="font-family: sans-serif;">
      <table style="text-align: center;" border="1">
        <tr style="font-weight: bold; font-size: 150%;">
          <td width="30%">Default mode</td>
          <td width="30%">Blue mode</td>
          <td width="30%">Red mode</td>
        </tr>
        <tr>
          <td>
            <p>
              <xsl:apply-templates mode="#default"
                select="/cars/manufacturer"/>
            </p>
          </td>
          <td>
            <p>
              <xsl:apply-templates mode="blue"
                select="/cars/manufacturer"/>
            </p>
          </td>
          <td>
            <p>
              <xsl:apply-templates mode="red"
```

```
                select="/cars/manufacturer"/>
            </p>
          </td>
        </tr>
      </table>
    </body>
  </html>
</xsl:template>

<xsl:template match="manufacturer">
  <div style="color: green; font-style: italic; font-size: 125%">
    <xsl:apply-templates select="car" mode="#current"/>
  </div>
</xsl:template>

<xsl:template match="manufacturer" mode="blue red">
  <div style="color: blue; font-weight: bold;">
    <xsl:apply-templates select="car" mode="#current"/>
  </div>
</xsl:template>

<xsl:template match="car" mode="#all">
  <xsl:value-of select="."/>
  <br/>
</xsl:template>

<xsl:template match="car" mode="red" priority="1">
  <div style="color: red; font-size: 125%; font-family: serif;">
    <xsl:value-of select="."/>
    <br/>
  </div>
</xsl:template>

</xsl:stylesheet>
```

In XSLT 2.0, the `mode` attribute on `<xsl:apply-templates>` can have the new values `#current` or `#default`. The `mode` attribute on `<xsl:template>` can have the new values `#all` or `#default`. Our sample stylesheet uses all of the new values:

- In the `match="/"` template, we use `<xsl:apply-templates>` with the modes `#default`, `blue`, and `red`. The only difference between the three `<xsl:apply-template>` elements is the `mode`.
- The first `match="manufacturer"` template doesn't have a `mode` attribute, so it applies to the default mode. When we use `<xsl:apply-templates mode="#default" ...>`, this template is the one that gets invoked.
- The second `match="manufacturer"` template has a `mode` of `blue red`. This template is invoked whenever the `mode` is `blue` or `red`.
- Both of the `match="manufacturer"` templates use `<xsl:apply-templates mode="#current" ...>` to process the `<car>` elements. The current `mode` is effectively passed as a parameter to the XSLT processor, telling it which template to apply.

- The first `match="car"` template is defined with `mode="#all"`, so it is the default template invoked by both of the `<xsl:apply-templates mode="#current" ...>` elements.
- The final `match="car"` template is defined with `mode="red"`. To make sure it is a better match than the `mode="#all"` template, we add the attribute `priority="1"`. Without this, Saxon displays an "Ambiguous rule match" warning, even though it invokes the `mode="red"` template when the `red` mode is in effect. The Altova XML engine doesn't issue any warnings—it simply invokes the template with `mode="red"`.

Starting with the `match="/"` template, our stylesheet called a variety of templates with three different modes to generate the output document. The three modes formatted the same information in three different ways. This stylesheet produces the same HTML document as the previous stylesheet.

See Also

The description of the `<xsl:template>` element.

<xsl:attribute>

Allows you to create an attribute in the output document. Using the `<xsl:attribute>` instruction allows you to determine the name or content of an attribute at runtime. You can build the attribute's name or value from parts of the input document, hardcoded text, values returned by functions, global variables, and any other value you can access from your stylesheet. You can also use `<xsl:if>` to determine whether the attribute should be created at all.

Category

Instruction.

Required Attribute

`name`

The `name` attribute defines the name of the attribute created by the `<xsl:attribute>` element. (No matter how you try to say this, talking about the attributes of the `<xsl:attribute>` element is confusing, isn't it?)

Optional Attributes

`namespace`

The `namespace` attribute defines the namespace URI that should be used for this attribute in the output document. You don't have control over the namespace prefix used; the only thing you specify with the `namespace` attribute is the namespace's URI.

[2.0] `select`
: An XPath expression that defines the content of this attribute. If the `<xsl:attribute>` element has a `select` attribute, the element must be empty.

[2.0] `separator`
: Defines the characters that separate multiple values generated by the `<xsl:attribute>` instruction. If `<xsl:attribute>` has a `select` attribute, the default value is a single space (`#x20`). Without a `select` attribute, the default value is a zero-length string (`""`). The `separator` attribute overrides the default value whether `select` is used or not.

[2.0 – Schema] `type`
: Defines the datatype of this attribute. The datatype can be any of the built-in datatypes, or it can be a datatype defined in a schema if you have a schema-aware XSLT 2.0 processor.

  The `type` and `validation` attributes are mutually exclusive.

[2.0 – Schema] `validation`
: Defines how the value of the new attribute will be validated. The `validation` attribute has four values: `strict`, `lax`, `preserve`, or `strip`. Because we're creating an attribute, as opposed to an element or a result document, some of the details of the `validation` attribute don't apply.

  The values `strip` and `preserve` have no effect. These two values define whether the type annotation of the attribute should be stripped from or preserved in the generated attribute. For an attribute, no validation is done, and the type annotation of the attribute is `xs:untypedAtomic`.

  `validation="strict"` means that the XSLT processor looks in all the declared schemas for a global attribute declaration (`<xs:attribute>`) with the same name as this attribute. It is a fatal error if the processor can't find a matching `<xs:attribute>`. Assuming the processor finds the declaration of the attribute, it validates the generated value against the attribute's declaration.

  The final value, `validation="lax"`, works just like `validation="strict"`, except that no error occurs if the processor can't find the declaration of the attribute in any declared schemas. In that case, the type annotation of the attribute is `xs:untypedAtomic`.

  The `validation` and `type` attributes are mutually exclusive.

Content

You can build the contents of an attribute with any elements that generate a *[1.0]* node-set or a *[2.0]* sequence. The `<xsl:choose>`, `<xsl:text>`, `<xsl:value-of>`, and *[2.0]* `<xsl:sequence>` elements all fit this description.

Appears in

`<xsl:attribute>` appears inside a template.

Defined in

[1.0] XSLT section 7.1.3, "Creating Attributes with `xsl:attribute`."

[2.0] XSLT section 11.3, "Creating Attribute Nodes Using `xsl:attribute`."

Example

For this example, we want to create an HTML table from the following XML document:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

We'll create a table that has each `<listitem>` in a separate row in the right column of the table, and a single cell with `rowspan` equal to the number of `<listitem>` elements in the XML document on the left. Clearly we can't hardcode a value for the `rowspan` attribute because the number of `<listitem>`s can change. This stylesheet uses `<xsl:attribute>` to do what we want:

```
<?xml version="1.0"?>
<!-- attribute1.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title><xsl:value-of select="list/title"/></title>
      </head>
      <body style="font-family: sans-serif;">
        <xsl:apply-templates select="list"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="list">
    <table border="1" cellpadding="5" cellspacing="5">
      <tr>
        <td style="background: black; color: white;
                   font-weight: bold; font-size: 125%;"
                   width="100" align="right">
          <xsl:if test="count(listitem) > 1">
            <xsl:attribute name="rowspan">
              <xsl:value-of select="count(listitem)"/>
            </xsl:attribute>
```

```
          </xsl:if>
          <xsl:value-of select="title"/>
        </td>
        <td>
          <xsl:value-of select="listitem[1]"/>
        </td>
      </tr>
      <xsl:for-each select="listitem[position() > 1]">
        <tr>
          <td>
            <xsl:value-of select="."/>
          </td>
        </tr>
      </xsl:for-each>
    </table>
  </xsl:template>

</xsl:stylesheet>
```

Here is the generated HTML document:

```
<!-- attribute.html -->
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Albums I've bought recently:</title>
   </head>
   <body style="font-family: sans-serif;">
      <table border="1" cellpadding="5" cellspacing="5">
         <tr>
            <td style="background: black; color: white;
                       font-weight: bold; font-size: 125%;"
                       width="100" align="right" rowspan="7">
              Albums I've bought recently:</td>
            <td>The Sacred Art of Dub</td>
         </tr>
         <tr>
            <td>Only the Poor Man Feel It</td>
         </tr>
         ...
      </table>
   </body>
</html>
```

Notice that the `<td>` element had several attributes hardcoded on it; those attributes are combined with the attribute we created with `<xsl:attribute>`. You can have as many `<xsl:attribute>` elements as you want, but all the `<xsl:attribute>` elements must appear before anything else in the template. Figure A-3 shows how our generated HTML document looks.

Be aware that in this instance, we could have used an attribute-value template. You could generate the value of the `rowspan` attribute like this:

```
<td bgcolor="black" rowspan="{count(listitem)}"
  width="100" align="right">
```

Figure A-3. Document with generated attributes

The expression in curly braces ({}) is evaluated and replaced with whatever its value happens to be. In this case, `count(listitem)` returns the number `7`, which becomes the value of the `rowspan` attribute. The one difference here is that the `rowspan` is generated when there is only one `<listitem>` element.

[2.0] Next we'll look at a brief example that illustrates the new `select` and `separator` attributes. This short stylesheet uses an XPath 2.0 range expression (a `"to"` expression) to assign a sequence of values to the `example` attribute:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- attribute2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" omit-xml-declaration="yes"/>

  <xsl:template match="/">
    <sampledoc>
      <xsl:attribute name="example" select="1 to 7" separator=", "/>
    </sampledoc>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates the following very short document:

```
<sampledoc example="1, 2, 3, 4, 5, 6, 7"/>
```

The XSLT 2.0 `separator` attribute means you don't have to write logic like this:

```
<xsl:for-each>
  <xsl:value-of select="."/>
  <xsl:if test="position() != last()">
    <xsl:text>, </xsl:text>
  </xsl:if>
</xsl:for-each>
```

[2.0 – Schema] For a final example, we'll use the schema-aware attributes of <xsl:attribute>. Our stylesheet contains an imported schema that defines a new datatype; we'll use <xsl:attribute> to create attributes of that datatype. Here's the stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- attribute3.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:zip="http://www.oreilly.com/xslt/zip"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs">

  <xsl:output method="xml" indent="yes"/>

  <xsl:import-schema namespace="http://www.oreilly.com/xslt/zip">
    <xsd:schema
      xmlns="http://www.oreilly.com/xslt/zip"
      targetNamespace="http://www.oreilly.com/xslt/zip"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">

      <xsd:simpleType name="zipcode">
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="[0-9]{5}(-[0-9]{4})?"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:schema>
  </xsl:import-schema>

  <xsl:template match="/">
    <postcodes>
      <xsl:for-each select="postcodes/postcode">
        <postcode>
          <xsl:choose>
            <xsl:when test=". castable as zip:zipcode">
              <xsl:attribute name="zip:zip" type="zip:zipcode">
                <xsl:value-of select=". cast as zip:zipcode"/>
              </xsl:attribute>
            </xsl:when>
            <xsl:otherwise>
              <xsl:attribute name="other" type="xs:string">
                <xsl:value-of select="."/>
              </xsl:attribute>
            </xsl:otherwise>
          </xsl:choose>
        </postcode>
      </xsl:for-each>
    </postcodes>
  </xsl:template>

</xsl:stylesheet>
```

We'll use this stylesheet to process this document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- postcodes.xml -->
<postcodes>
```

```
  <postcode>3S8 EOX</postcode>
  <postcode>37174</postcode>
  <postcode>NSW 3829</postcode>
  <postcode>27516</postcode>
</postcodes>
```

The stylesheet uses the XPath `castable as` operator to determine whether a value from the source document can be cast as a `zip:zipcode` value. If the value can be cast, we create a new attribute with the datatype `zip:zipcode`. Here are the results:

```
<?xml version="1.0" encoding="UTF-8"?>
<postcodes xmlns:zip="http://www.oreilly.com/xslt/zip">
   <postcode other="3S8 EOX"/>
   <postcode zip:zip="37174"/>
   <postcode other="NSW 3829"/>
   <postcode zip:zip="27516"/>
</postcodes>
```

Two of the generated attributes are namespace-qualified and contain values of the appropriate datatype.

<xsl:attribute-set>

Allows you to define a group of attributes for the output. You can then reference the entire attribute set with its name, rather than create all attributes individually.

Category

Top-level element.

Required Attribute

`name`
: Defines the name of this attribute set.

Optional Attribute

`use-attribute-sets`
: Lists one or more attribute sets that should be used by this attribute set. If you specify more than one set, separate their names with whitespace characters. You can use this attribute to embed other `<xsl:attribute-set>`s in this one, but be aware that an `<xsl:attribute-set>` that directly or indirectly embeds itself results in an error. In other words, if attribute set `A` embeds attribute set `B`, and attribute set `B` embeds attribute set `C`, and attribute set `C` embeds attribute set `A`, the XSLT processor signals an error.

Content

One or more `<xsl:attribute>` elements.

Appears in

`<xsl:attribute-set>` is a top-level element and can only appear as a child of `<xsl:stylesheet>`.

Defined in

[1.0] XSLT section 7.1.4, "Named Attribute Sets."

[2.0] XSLT section 10.2, "Named Attribute Sets."

Example

We'll use a stylesheet with three `<xsl:attribute-set>`s. The first set defines a `style` attribute with some CSS properties, the second defines a couple of HTML `<table>` attributes and uses the first set, while the third defines a traditional HTML element and a CSS `style` attribute.

```
<?xml version="1.0"?>
<!-- attribute-set.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:attribute-set name="bold-table">
    <xsl:attribute name="style">
      font-weight: bold;
    </xsl:attribute>
  </xsl:attribute-set>

  <xsl:attribute-set name="spacious-table"
    use-attribute-sets="bold-table">
    <xsl:attribute name="cellpadding">8</xsl:attribute>
    <xsl:attribute name="cellspacing">8</xsl:attribute>
  </xsl:attribute-set>

  <xsl:attribute-set name="reverse-table">
    <xsl:attribute name="bgcolor">black</xsl:attribute>
    <xsl:attribute name="style">color: white;</xsl:attribute>
  </xsl:attribute-set>

  <xsl:template match="/">
    <html>
      <head>
        <title><xsl:value-of select="/list/title"/></title>
      </head>
      <body style="font-family: sans-serif;">
        <xsl:apply-templates select="*"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="list">
    <h1><xsl:value-of select="title"/></h1>
    <table xsl:use-attribute-sets="spacious-table" border="2">
      <xsl:for-each select="listitem">
```

```
        <tr>
          <td xsl:use-attribute-sets="reverse-table">
            <xsl:value-of select="."/>
          </td>
        </tr>
      </xsl:for-each>
    </table>
    <h1>Here's the same table with different attribute sets:</h1>
    <table border="2" xsl:use-attribute-sets="bold-table">
      <xsl:for-each select="listitem">
        <tr>
          <td>
            <xsl:value-of select="."/>
          </td>
        </tr>
      </xsl:for-each>
    </table>
  </xsl:template>

</xsl:stylesheet>
```

Notice that you have to namespace-qualify the `xsl:use-attribute-sets` attribute so that the XSLT processor will handle it. If you just use `use-attribute-sets`, the XSLT processor copies that attribute to the HTML output without processing it at all.

Our stylesheet creates two tables from a list of items, using a different set of named attribute sets each time. We'll apply our stylesheet to one of our usual XML documents:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list>
  <title>A few of my favorite albums</title>
  <listitem>A Love Supreme</listitem>
  <listitem>Beat Crazy</listitem>
  <listitem>You Could Have it So Much Better</listitem>
  <listitem>Kind of Blue</listitem>
  <listitem>London Calling</listitem>
  <listitem>Remain in Light</listitem>
  <listitem>The Joshua Tree</listitem>
  <listitem>The Indestructible Beat of Soweto</listitem>
</list>
```

Our stylesheet produces this HTML:

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Albums I've bought recently:</title>
   </head>
   <body>
      <h1>Albums I've bought recently:</h1>
      <table style="font-weight: bold; font-family: sans-serif;"
                  cellpadding="8" cellspacing="8" border="2">
        <tr>
           <td bgcolor="black" style="color: white;">The Sacred Art of Dub</td>
        </tr>
```

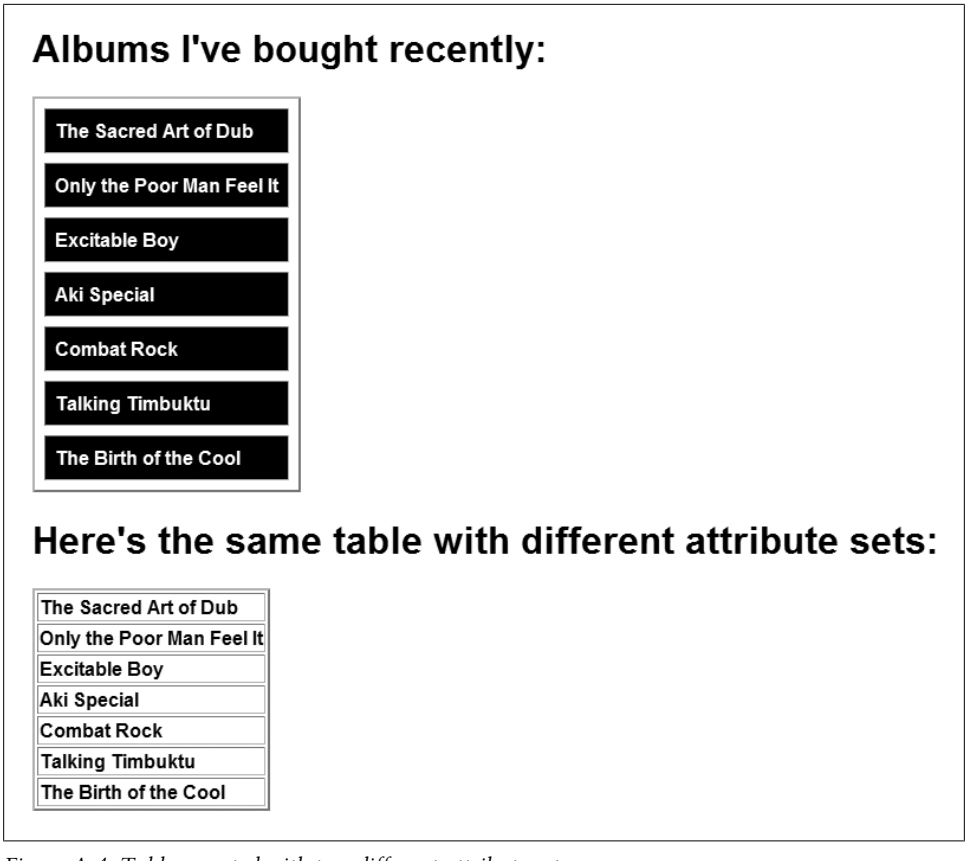


Figure A-4. Tables created with two different attribute sets

```
            ...
            <tr>
               <td bgcolor="black" style="color: white;">The Birth of the Cool</td>
            </tr>
         </table>
         <h1>Here's the same table with different attribute sets:</h1>
         <table style="font-weight: bold; font-family: sans-serif;" border="2">
            <tr>
               <td>The Sacred Art of Dub</td>
            </tr>
            ...
            <tr>
               <td>The Birth of the Cool</td>
            </tr>
         </table>
      </body>
   </html>
```

The two tables look like Figure A-4.

Be aware that any attribute coded on an element takes precedence over an attribute added to that element by `xsl:use-attribute-sets`. For example, if we have this element:

```
<table cellpadding="3" xsl:use-attribute-sets="spacious-table" ...>
```

The `cellpadding` attribute in the `spacious-table` attribute set does not apply. The XSLT processor generates this `<table>` element:

```
<table cellpadding="3" cellspacing="8" ...>
```

<xsl:call-template>

Invokes a particular template by name. If you have output that you need to generate often, you can put the markup that creates that output into a named template and invoke that template whenever you want.

Category

Instruction.

Required Attribute

`name`
: The name of the template you're invoking.

Optional Attributes

None.

Content

This element can contain any number of optional `<xsl:with-param>` elements.

Appears in

`<xsl:call-template>` appears inside a template.

Defined in

[1.0] XSLT section 6, "Named Templates."

[2.0] XSLT section 10.1, "Named Templates."

Example

Invoking named templates with `<xsl:call-template>` is a great way to create modular stylesheets. We'll use named templates to format our document of chocolate sales:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
```

```
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

Here is our stylesheet with named templates:

```
<?xml version="1.0"?>
<!-- call-template.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="report">
    <html>
      <head>
        <xsl:call-template name="report-title">
          <xsl:with-param name="in-heading" select="true()"/>
          <xsl:with-param name="title" select="title"/>
        </xsl:call-template>
      </head>
      <body>
        <xsl:call-template name="insert-header"/>
        <xsl:call-template name="report-title">
          <xsl:with-param name="in-heading" select="false()"/>
          <xsl:with-param name="title" select="title"/>
        </xsl:call-template>
        <table cellpadding="5">
          <xsl:call-template name="table-heading"/>
          <xsl:apply-templates select="brand"/>
        </table>
        <xsl:call-template name="insert-footer"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template name="insert-header">
    <p style="font-size: 75%; font-style: italic;">
      This confidential report is the property of DougCo, Inc.
    </p>
    <hr/>
```

```
  </xsl:template>

  <xsl:template name="report-title">
    <xsl:param name="in-heading"/>
    <xsl:param name="title"/>
    <xsl:choose>
      <xsl:when test="$in-heading">
        <title><xsl:value-of select="$title"/></title>
      </xsl:when>
      <xsl:otherwise>
        <h1><xsl:value-of select="$title"/></h1>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

  <xsl:template name="table-heading">
    <tr>
      <td style="background: black; color: white; font-weight: bold;">
        Brand</td>
      <td style="background: black; color: white; font-weight: bold;">
        Sales</td>
    </tr>
  </xsl:template>

  <xsl:template name="insert-footer">
    <hr/>
    <p style="font-size: 75%; font-style: italic;">
      © Copyright 2008, DougCo, Inc.
    </p>
  </xsl:template>

  <xsl:template match="brand">
    <tr>
      <td><xsl:value-of select="name"/></td>
      <td><xsl:value-of select="units"/></td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

Our stylesheet uses four different named templates. The templates `insert-header`, `table-heading`, and `insert-footer` make it easy to insert commonly used markup into the output. In practice, templates such as this would be imported from another stylesheet. If dozens of reports, web pages, or other documents used the same header and footer, we could store that markup in one place and add it wherever we needed with a simple `<xsl:call-template>` instruction.

The other named template, `insert-title`, uses two parameters. The first parameter tells the named template whether we're in the `<head>` section of the HTML document; the second parameter is the name of the report title. Depending on the value of `in-heading`, the template generates a `<title>` element or a `<h1>` element.

Here is the HTML file generated by the stylesheet:

```
<html>
  <head>
    <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Chocolate bar sales</title>
  </head>
  <body>
    <p style="font-size: 75%; font-style: italic;">
      This confidential report is the property of DougCo, Inc.
    </p>
    <hr>
    <h1>Chocolate bar sales</h1>
    <table cellpadding="5">
      <tr>
        <td style="background: black; color: white; font-weight: bold;">
          Brand
        </td>
        <td style="background: black; color: white; font-weight: bold;">
          Sales
        </td>
      </tr>
      <tr>
        <td>Lindt</td><td>27408</td>
      </tr>
      <tr>
        <td>Callebaut</td><td>8203</td>
      </tr>
      <tr>
        <td>Valrhona</td><td>22101</td>
      </tr>
      <tr>
        <td>Perugina</td><td>14336</td>
      </tr>
      <tr>
        <td>Ghirardelli</td><td>19268</td>
      </tr>
    </table>
    <hr>
    <p style="font-size: 75%; font-style: italic;">
      &copy; Copyright 2007, DougCo, Inc.
    </p>
  </body>
</html>
```

The results look like Figure A-5.

[2.0] In XSLT 2.0, there are changes to the rules for passing parameters to templates:

- In XSLT 1.0, you could pass as many parameters as you wanted to a template; any extra parameters (parameters not defined in the called template) were ignored. In XSLT 2.0, this is a fatal error.
- In XSLT 2.0, the `<xsl:param>` element has a `required` attribute. It is a fatal error to call a template without passing values for all of the required parameters.
- XSLT 2.0 introduces the concept of tunnel parameters. See the section "Tunnel parameters" in Chapter 5 for more information on tunnel parameters.

Figure A-5. HTML document generated with named templates

[2.0] <xsl:character-map>

Defines a set of characters, each of which should be replaced by a string of characters. This allows you to put nonstandard characters in the values of elements and attributes. An `<xsl:character-map>` works much like an XML `<!ENTITY>` declaration. Use a `<xsl:character-map>` in place of the `disable-output-escaping` attributes of `<xsl:text>` and `<xsl:value-of>` defined in XSLT 1.0. (Using the `disable-output-escaping` attribute is deprecated in XSLT 2.0.)

Category

Declaration.

Required Attribute

`name`
: The name of this character map.

Optional Attribute

`use-character-maps`
: The space-separated names of any character maps included in this character map. As you would expect, it is a fatal error if a character map includes itself, directly or indirectly. It is also a fatal error if a character map attempts to include a character map that does not exist.

Content

Zero or more <xsl:output-character> elements.

Appears in

<xsl:character-map> appears as a child of the <xsl:stylesheet> element.

Defined in

XSLT section 20, "Serialization."

Example

We'll define a simple example that has a couple of useful functions. First of all, we'll create a character mapping that replaces tab characters (	) with two spaces; tab characters are often displayed as eight characters wide, which can cause problems when displaying indented code listings. We'll also create a couple of graphics that should be displayed in place of certain characters. Here's the XML source we'll use:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- special-characters.xml -->
<char-test>
  <tabs>public class HelloWorld {
        public static void main(String[] args) {
                System.out.println("Hello, World!");
        }
}</tabs>
  <special-char>&#x2780;</special-char>
  <special-char>&#x2781;</special-char>
</char-test>
```

Our stylesheet replaces the tab characters with two spaces, and it replaces the first two characters with graphics. (The Unicode characters ➀ through ➉ are the circled numbers 1 through 10 in a sans-serif fonti.e., ① through ⑩.) Here's the stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- character-map1.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" use-character-maps="sample"/>

  <xsl:character-map name="sample" use-character-maps="circles">
    <xsl:output-character character="&#x9;" string="  "/>
  </xsl:character-map>

  <xsl:character-map name="circles">
    <xsl:output-character character="&#x2780;"
      string="&lt;img src='images/circle1.gif'
      width='28' height='28'/&gt;"/>
    <xsl:output-character character="&#x2781;"
      string="&lt;img src='images/circle2.gif'
      width='28' height='28'/&gt;"/>
  </xsl:character-map>
```

```
  <xsl:template match="char-test">
    <html>
      <head>
        <title>A test of some special characters</title>
      </head>
      <body style="font-family: sans-serif;">
        <h1>A test of some special characters</h1>
        <xsl:apply-templates select="*"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="tabs">
    <pre style="font-size: 150%; font-weight: bold;">
      <xsl:value-of select="."/>
    </pre>
  </xsl:template>

  <xsl:template match="special-char">
    <p style="font-size: 200%;">
      <xsl:text>Here's a special character: </xsl:text>
      <xsl:value-of select="."/>
    </p>
  </xsl:template>

</xsl:stylesheet>
```

Notice that our stylesheet simply outputs the value of the `<tabs>` or `<special-char>` elements. The `<xsl:output>` element and its `use-character-maps="sample"` attribute take care of transforming the characters for us. The `<xsl:character-map>` named `sample` defines a single replacement, and it also uses `<xsl:character-map name="circles">`. The combination of defining a character map and referencing it on the `<xsl:output>` element makes everything happen.

Also notice that the substitutions in the `circles` character map replace a character with HTML markup—an `<img>` element. The HTML document looks like this:

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>A test of some special characters</title>
   </head>
   <body style="font-family: sans-serif;">
      <h1>A test of some special characters</h1>
      <pre style="font-size: 150%;
      font-weight: bold;">public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}</pre><p style="font-size: 200%;">
      Here's a special character:
      <img src='images/circle1.gif' width='28' height='28'/></p>
    <p style="font-size: 200%;">
      Here's a special character:
      <img src='images/circle2.gif' width='28' height='28'/></p>
```

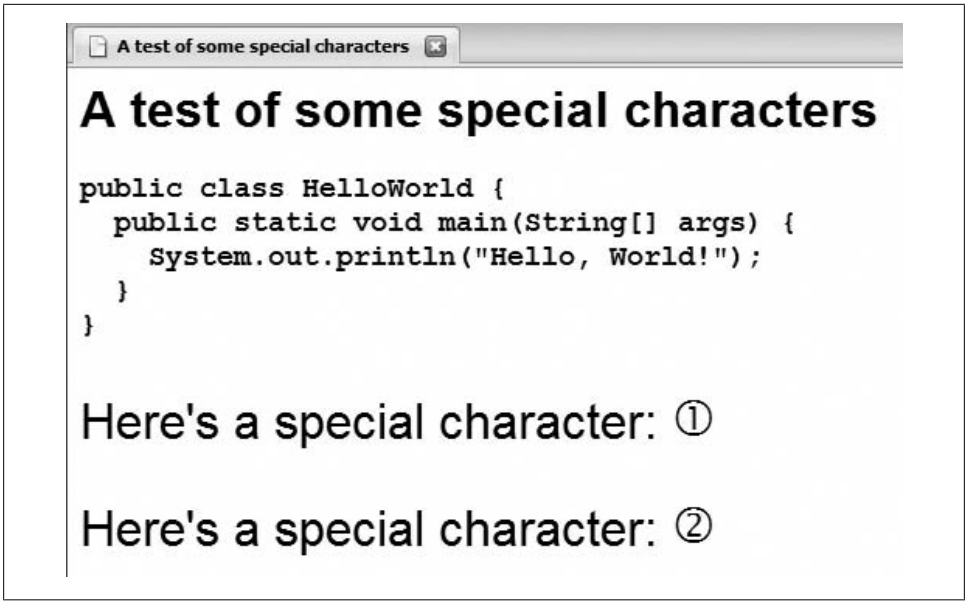

A test of some special characters
A test of some special characters
public class HelloWorld {
public static void main(String[] args) {
System.out.println("Hello, World!");
}
}
Here's a special character: ①
Here's a special character: ②

Figure A-6. HTML generated with character maps

```
    </body>
</html>
```

When displayed in a browser, the circled numbers appear, as shown in Figure A-6.

Another common use of `<xsl:character-map>` is to convert a newline character (`
`) to a carriage return and line feed (`
`), but that's not a visually compelling demo. If you need to do that, the `<xsl:output-character>` element looks like this:

```
<xsl:output-character character="&#xA;" string="&#xD;&#xA;"/>
```

Finally, the XSLT 2.0 spec mentions creating Java Server Pages (JSPs) as an example. JSPs use angle brackets in a way that is not valid XML or HTML. We can define a character map that looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- character-map2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" use-character-maps="jsp"/>

  <xsl:character-map name="jsp">
    <xsl:output-character character="«" string="&lt;%="/>
    <xsl:output-character character="»" string="%&gt;"/>
  </xsl:character-map>

  <xsl:template match="jsp-test">
    <html>
      <head>
        <title>A test of some special characters</title>
```

```
        </head>
        <body style="font-family: sans-serif;">
          <h1>Generating a .jsp page</h1>
          <p>Here's the value of a JSP function:
            <xsl:apply-templates select="text()"/>
          </p>
        </body>
      </html>
    </xsl:template>

</xsl:stylesheet>
```

Using this XML input file:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- jsp-test.xml -->
<jsp-test>« new java.util.Date() »</jsp-test>
```

generates these results:

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>A test of some special characters</title>
   </head>
   <body style="font-family: sans-serif;">
      <h1>Generating a .jsp page</h1>
      <p>Here's the value of a JSP function:
         <%= new java.util.Date() %>
      </p>
   </body>
</html>
```

<xsl:choose>

The `<xsl:choose>` element is XSLT's construct for if-then-else processing.

Category

Instruction.

Required Attributes

None.

Optional Attributes

None.

Content

Contains one or more `<xsl:when>` elements. It can also contain a single `<xsl:otherwise>` element. If it is present, the `<xsl:otherwise>` element must be the last element inside `<xsl:choose>`.

Appears in

<xsl:choose> appears inside a template.

Defined in

[1.0] XSLT section 9.2, "Conditional Processing with xsl:choose."

[2.0] XSLT section 8.2, "Conditional Processing with xsl:choose."

Example

Here's an example that uses <xsl:choose> to select the background color for the rows of an HTML table. We cycle among four different values, using <xsl:choose> to determine the value of the style attribute in the generated HTML document. Here's the XML document we'll use:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

And here's our stylesheet:

```
<?xml version="1.0"?>
<!-- choose.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>
          <xsl:value-of select="list/title"/>
        </title>
      </head>
      <body style="font-family: sans-serif; color: white;">
        <h1 style="color: black;">
          <xsl:value-of select="list/title"/>
        </h1>
        <table border="1" cellpadding="5"
          style="font-weight: bold;">
          <xsl:for-each select="list/listitem">
            <tr>
              <td>
                <xsl:attribute name="style">
                  <xsl:choose>
                    <xsl:when test="position() mod 4 = 0">
```

```
                    <xsl:text>background: yellow; color: black;</xsl:text>
                  </xsl:when>
                  <xsl:when test="position() mod 4 = 1">
                    <xsl:text>background: blue;</xsl:text>
                  </xsl:when>
                  <xsl:when test="position() mod 4 = 2">
                    <xsl:text>background: white; color: black;</xsl:text>
                  </xsl:when>
                  <xsl:otherwise>
                    <xsl:text>background: black;</xsl:text>
                  </xsl:otherwise>
                </xsl:choose>
              </xsl:attribute>
              <xsl:value-of select="."/>
            </td>
          </tr>
        </xsl:for-each>
      </table>
    </body>
  </html>
 </xsl:template>

</xsl:stylesheet>
```

We use `<xsl:choose>` to generate the `style` attribute of each generated `<td>` element. Here's the generated HTML document, which cycles through the various background colors:

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Albums I've bought recently:</title>
   </head>
   <body style="font-family: sans-serif; color: white;">
      <h1 style="color: black;">Albums I've bought recently:</h1>
      <table border="1" cellpadding="5" style="font-weight: bold;">
         <tr>
            <td style="background: blue;">The Sacred Art of Dub</td>
         </tr>
         ...
         <tr>
            <td style="background: black;">The Birth of the Cool</td>
         </tr>
      </table>
   </body>
</html>
```

When rendered, our HTML document looks like Figure A-7.

<xsl:comment>

Allows you to create a comment in the output document. Comments are sometimes used to add legal notices, disclaimers, or information about when the output document was created.

Figure A-7. Using <xsl:choose> to cycle among background colors

Another useful application of the `<xsl:comment>` element is generating CSS definitions or JavaScript code in an HTML document.

Category

Instruction.

Required Attributes

None.

Optional Attribute

[2.0] `select`

An XPath expression that generates content for the comment. If this attribute is not present, the contents of the `<xsl:comment>` element are used instead. If the `select` attribute is not present and the `<xsl:comment>` element is empty, an empty comment is generated. It is a fatal error for an `<xsl:comment>` element to have a `select` attribute and contain content.

Content

A *[1.0]* node-set or *[2.0]* sequence constructor.

Appears in

`<xsl:comment>` appears in a template.

Defined in

[1.0] XSLT section 7.4, "Creating Comments."

[2.0] XSLT section 11.8, "Creating Comments."

Example

Here's a stylesheet that generates a comment to define CSS styles in an HTML document:

```
<?xml version="1.0"?>
<!-- comment.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>XSLT and CSS Demo</title>
        <style>
          <xsl:comment>
            p.big   {font-size: 125%; font-weight: bold;}
            p.odd   {color: purple; font-weight: bold;}
            p.even  {color: blue; font-style: italic; font-weight: bold;}
          </xsl:comment>
        </style>
      </head>
      <body style="font-family: sans-serif;">
        <xsl:apply-templates select="list/title"/>
        <xsl:apply-templates select="list/listitem"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="title">
    <p class="big"><xsl:value-of select="."/></p>
  </xsl:template>

  <xsl:template match="listitem">
    <xsl:choose>
      <xsl:when test="position() mod 2">
        <p class="odd"><xsl:value-of select="."/></p>
      </xsl:when>
      <xsl:otherwise>
        <p class="even"><xsl:value-of select="."/></p>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet creates three CSS styles inside an HTML comment. We'll apply the stylesheet to this document:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
```

Figure A-8. A document with CSS properties defined in comment nodes

```
    <listitem>The Sacred Art of Dub</listitem>
    <listitem>Only the Poor Man Feel It</listitem>
    <listitem>Excitable Boy</listitem>
    <listitem xml:lang="sw">Aki Special</listitem>
    <listitem xml:lang="en-gb">Combat Rock</listitem>
    <listitem xml:lang="zu">Talking Timbuktu</listitem>
    <listitem xml:lang="jz">The Birth of the Cool</listitem>
  </list>
```

The stylesheet applies one CSS style to the `<title>` element and alternates between two CSS styles for the `<listitem>`s. Here's the generated HTML:

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>XSLT and CSS Demo</title><style>
         <!--
            p.big      {font-size: 125%; font-weight: bold;}
            p.odd      {color: purple; font-weight: bold;}
            p.even     {color: blue; font-style: italic; font-weight: bold;}
          --></style></head>
   <body style="font-family: sans-serif;">
      <p class="big">Albums I've bought recently:</p>
      <p class="odd">The Sacred Art of Dub</p>
      <p class="even">Only the Poor Man Feel It</p>
      <p class="odd">Excitable Boy</p>
      <p class="even">Aki Special</p>
      <p class="odd">Combat Rock</p>
      <p class="even">Talking Timbuktu</p>
      <p class="odd">The Birth of the Cool</p>
   </body>
</html>
```

When rendered, the document looks like Figure A-8.

[2.0] To use the `select` attribute added in XSLT 2.0, we could add this `<xsl:comment>` element to our stylesheet:

```
<xsl:comment select="concat('The second album is ', list/listitem[2])"/>
```

This generates the following comment in the HTML output:

```
<!--The second album is Only the Poor Man Feel It-->
```

If you're using `<xsl:comment>` to generate JavaScript code, end your code with a JavaScript comment, like this:

```
<SCRIPT type="text/javascript">
<xsl:comment> // Hide this code from older browsers
  function iOver(image)
  {
    if (browser="N3") document[image].src=eval(image + "over.src");
  }
  // </xsl:comment>
</SCRIPT>
```

Putting the two slashes in the last line creates a JavaScript comment. That means the JavaScript interpreter ignores the `-->` that appears at the end of the comment.

<xsl:copy>

By default, `<xsl:copy>` makes a shallow copy of a node. The `<xsl:copy>` instruction copies only the current node and its namespace nodes; attribute or child nodes are not copied. `<xsl:copy>` gives you fine-grained control over the copying process, but it requires you to do more work. You can use `<xsl:copy>` to copy any kind of node, including comment or attribute nfodes.

Category

Instruction.

Required Attributes

None.

Optional Attributes

`use-attribute-sets`
: Lists one or more attribute sets that should be used by this element. If you specify more than one attribute set, separate their names with whitespace characters. See the description of the `<xsl:attribute-set>` element for more information.

[2.0] `copy-namespaces`
: Defines whether namespaces should be copied. This applies only when copying an element node. Allowed values are `yes` (the default) and `no`.

[2.0] `inherit-namespaces`
: Defines whether this element and its children inherit the current namespace nodes. Valid values are `yes` (the default) and `no`.

[2.0 – Schema] `type`
: Defines the datatype of the copied element. The datatype can be any of the built-in datatypes, or it can be a datatype defined in a schema if you have a schema-aware XSLT 2.0 processor.

  The `type` and `validation` attributes are mutually exclusive.

[2.0 – Schema] `validation`
: Defines how the value of the copied node will be validated. The `validation` attribute has four values: `strict`, `lax`, `preserve`, or `strip`.

  `validation="strict"` means that the XSLT processor looks in all the declared schemas for an attribute or element declaration (`<xs:attribute>` or `<xs:element>`) with the same name as the copied node. It is a fatal error if the processor can't find a matching declaration. Assuming the processor finds the declaration of the node, it validates the copied node's value against its declaration.

  `validation="lax"` works just like `validation="strict"`, except that no error occurs if the processor can't find the declaration of the copied node in any of the declared schemas. In that case, the type annotation of the element is `xs:untyped`.

  The effects of `validation="preserve"` depend on the kind of node being copied. If the copied node is an attribute, its type annotation is preserved. For element nodes, the copied node will have the type annotation of `xs:anyType`. The type annotations of any nodes contained by the copied element are preserved. No schema validation is done.

  `validation="strip"` replaces the type annotation of the copied attribute and element nodes with `xs:untypedAtomic` and `xs:untyped`, respectively. Any attribute and element nodes contained in the copied node have their type annotations replaced with `xs:untypedAtomic` and `xs:untyped` as well.

  The `validation` and `type` attributes are mutually exclusive.

Content

An XSLT template.

Appears in

`<xsl:copy>` appears in a template.

Defined in

[1.0] XSLT section 7.5, "Copying."

[2.0] XSLT section 11.9.1, "Shallow Copying."

Example

We'll demonstrate <xsl:copy> with an example that copies an element to the result tree. Our first stylesheet simply copies the document element:

```
<?xml version="1.0"?>
<!-- copy1.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml"/>

  <xsl:template match="*">
    <xsl:copy/>
  </xsl:template>

</xsl:stylesheet>
```

We'll test our stylesheet with the following XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

We don't do anything to copy any of the children or attributes of the document element, so our results are very short:

```
<?xml version="1.0" encoding="UTF-8"?><list/>
```

To get more results, we need to copy the children of each element. Here is our stylesheet:

```
<?xml version="1.0"?>
<!-- copy2.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml"/>

  <xsl:template match="*">
    <xsl:copy>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
<?xml version="1.0" encoding="UTF-8"?><list>
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem>Aki Special</listitem>
  <listitem>Combat Rock</listitem>
  <listitem>Talking Timbuktu</listitem>
  <listitem>The Birth of the Cool</listitem>
</list>
```

Although we have more results, we still don't have any attributes in the result document. The `<xsl:copy>` does a shallow copy, which gives you complete control over the output (unlike `<xsl:copy-of>`). The downside of having this control is that you must explicitly specify any child nodes or attribute nodes you want to copy. (Because attribute nodes aren't considered children of their parent element, we would have to specifically select and copy them.) This stylesheet doesn't copy comment nodes or processing instructions either; we would have to add more markup to handle them.

Compare this example to the example for the `<xsl:copy-of>` element.

The `<xsl:copy>` instruction is also useful inside a template that matches more than one element. We'll use our document of chocolate sales figures:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

Our stylesheet uses `<xsl:copy>` to copy just the `<brand>` and `<name>` elements:

```
<?xml version="1.0"?>
<!-- copy3.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
  <xsl:output method="xml"/>

  <xsl:template match="report">
    <brands>
      <xsl:apply-templates select="brand"/>
    </brands>
  </xsl:template>

  <xsl:template match="brand|name|units">
    <xsl:copy>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet creates a new `<brands>` element, and then copies all of the `<brand>` elements and their children. Here are the results:

```
<?xml version="1.0" encoding="UTF-8"?><report><brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand><brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand><brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand><brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand><brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand></report>
```

Notice that the text nodes of the selected elements are processed by the default stylesheet rule, which copies the text to the output.

<xsl:copy-of>

Creates a deep copy of a node.

Category

Instruction.

Required Attribute

`select`
: Contains an XPath expression that defines the nodes to be copied to the output document.

Optional Attributes

[2.0] `copy-namespaces`
: Defines whether namespaces should be copied. This applies only when copying an element node. Allowed values are `yes` (the default) and `no`.

[2.0 – Schema] `type`
: Defines the datatype of the copied node. The datatype can be any of the built-in datatypes, or it can be a datatype defined in a schema if you have a schema-aware XSLT 2.0 processor.

  The `type` and `validation` attributes are mutually exclusive.

[2.0 – Schema] `validation`
: Defines how the value of the copied node will be validated. The `validation` attribute has four values: `strict`, `lax`, `preserve`, or `strip`.

  `validation="strict"` means that the XSLT processor looks in all the declared schemas for a node declaration (`<xs:attribute>` or `<xs:element>`) with the same name as this node. It is a fatal error if the processor can't find a matching declaration. Assuming the processor finds the declaration of the attribute or element node, it validates the generated value against its declaration in the schema.

  `validation="lax"` works just like `validation="strict"`, except that no error occurs if the processor can't find the declaration of the node in any of the declared schemas. In that case, the type annotation of a copied attribute or element is `xs:untypedAtomic` or `xs:untyped`, respectively.

  The value `validation="preserve"` means that all the copied nodes will have their type annotations preserved.

  Finally, `validation="strip"` removes all of the type annotations from all of the copied nodes. The copied attribute nodes will have a type annotation of `xs:untypedAtomic` and the copied element nodes will have a type annotation of `xs:untyped`.

  The `validation` and `type` attributes are mutually exclusive.

Content

None. `<xsl:copy-of>` is an empty element.

Appears in

`<xsl:copy-of>` appears inside a template.

Defined in

[1.0] XSLT section 11.3, "Using Values of Variables and Parameters with `xsl:copy-of`."

[2.0] XSLT section 11.9.2, "Deep Copy."

Example

We'll demonstrate <xsl:copy-of> with a simple stylesheet that copies the input document to the result tree. Here is our stylesheet:

```
<?xml version="1.0"?>
<!-- copy-of.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml"/>

  <xsl:template match="/">
    <xsl:copy-of select="."/>
  </xsl:template>

</xsl:stylesheet>
```

We'll test our stylesheet with the following document:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

When we transform the XML document, the results are strikingly similar to the input document:

```
<?xml version="1.0" encoding="UTF-8"?><!-- albums.xml --><list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

The only differences between the two documents is that some whitespace has been removed and the stylesheet engine has added an `encoding` to the XML declaration. Semantically the two documents are identical. Compare this to the example for the <xsl:copy> element.

Be aware that this element works somewhat differently for XSLT 1.0 and 2.0:

[1.0]:

- If the `select` attribute identifies a result-tree fragment, the complete fragment is copied to the result tree.

- If `select` identifies a node-set, all nodes in the node-set are copied to the result tree in document order. Unlike `<xsl:copy>`, the node is copied in its entirety, including any namespace nodes, attribute nodes, and child nodes.
- If the `select` attribute identifies something other than a result-tree fragment or a node-set, it is converted to a string and inserted into the result tree.

[2.0]:

- If the `select` attribute identifies an element, that element and all of its descendants and attributes are copied to the output. By default, the element's namespace nodes are copied as well, although that can be changed with the `copy-namespaces` attribute.
- If the `select` attribute points to a document node, that document node and all of its descendants are copied to the output.
- All other types of nodes (attribute, namespace, text, comment, or processing instruction nodes) are copied to the output.
- Finally, atomic values are appended to the result sequence.

<xsl:decimal-format>

Defines a number format to be used when writing numeric values to the output document. If the `<decimal-format>` does not have a `name`, it is assumed to be the default number format used for all calls to the `format-number( )` function. On the other hand, if a number format is named, it can be referenced from the `format-number( )` function.

Category

Top-level element.

Required Attributes

None.

Optional Attributes

`name`
: Gives a name to this format.

`decimal-separator`
: Defines the character (usually either a period or comma) used as the decimal point. This character is used both in the format string and in the output. The default value is the period character (`.`).

`grouping-separator`
: Defines the character (usually either a period or comma) used as the thousands separator. This character is used both in the format string and in the output. The default value is the comma (`,`).

`infinity`
: Defines the string used to represent infinity. Be aware that XSLT's number facilities support both positive and negative infinity. This string is used only in the output. The default value is the string `"Infinity"`.

`minus-sign`
: Defines the character used as the minus sign. This character is used only in the output. The default value is the hyphen character (`-`, `-`).

`NaN`
: Defines the string displayed when the value to be formatted is not a number. This string is used only in the output; the default value is the string `"NaN"`.

`percent`
: Defines the character used as the percent sign. This character is used both in the format string and in the output. The default value is the percent sign (`%`).

`per-mille`
: Defines the character used as the per-mille sign. This character is used both in the format string and in the output. The default value is the Unicode per-mille character (‰, `‰`).

`zero-digit`
: Defines the character used for the digit zero. This character is used both in the format string and in the output. The default is the digit zero (`0`).

`digit`
: Defines the character used in the format string to stand for a digit. The default is the number sign character (`#`).

`pattern-separator`
: Defines the character used to separate the positive and negative subpatterns in a pattern. The default value is the semicolon (`;`). This character is used only in the format string.

Content

None. `<xsl:decimal-format>` is an empty element.

Appears in

`<xsl:decimal-format>` is a top-level element and can appear only as a child of `<xsl:stylesheet>`.

Defined in

[1.0] XSLT section 12.3, "Number Formatting."

[2.0] XSLT section 16.4.1, "Defining a Decimal Format."

Example

Here is a stylesheet that defines two `<decimal-format>`s:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- decimal-format.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

f  <xsl:decimal-format name="f1"
    decimal-separator=":"
    grouping-separator="/"/>

  <xsl:decimal-format name="f2"
    infinity="Really, really big"
    NaN="[not a number]"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the &lt;decimal-format&gt; element:</xsl:text>

    <xsl:text>&#xA;&#xA;    format-number(1528.3, '#/###:00', 'f1')=</xsl:text>
    <xsl:value-of select="format-number(1528.3, '#/###:00;-#/###:00', 'f1')"/>
    <xsl:text>&#xA;    format-number(1 div 0, '###,###.00', 'f2')=</xsl:text>
    <xsl:value-of select="format-number(1 div 0, '###,###.00', 'f2')"/>
    <xsl:text>&#xA;    format-number(blue div orange, '#.##', 'f2')=</xsl:text>
    <xsl:value-of select="format-number(blue div orange, '#.##', 'f2')"/>
    <xsl:text>&#xA;&#xA;************************************</xsl:text>
    <xsl:text>&#xA;Sales report for </xsl:text>
    <xsl:value-of select="/report/@month"/>
    <xsl:text>/</xsl:text>
    <xsl:value-of select="/report/@year"/>
    <xsl:text>&#xA;&#xA;</xsl:text>
    <xsl:variable name="totalSales" select="sum(/report/brand/units)"/>
    <xsl:for-each select="report/brand">
      <xsl:value-of select="name"/>
      <xsl:text>: &#xA;    </xsl:text>
      <xsl:value-of select="format-number(units, '##,###')"/>
      <xsl:text> bars sold, </xsl:text>
      <xsl:value-of select="format-number(units div $totalSales, '##%')"/>
      <xsl:text> of all sales.</xsl:text>
      <xsl:text>&#xA;</xsl:text>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
    <xsl:text>Total sales: </xsl:text>
    <xsl:value-of select="format-number($totalSales, '##,###')"/>
    <xsl:text> bars.&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

We'll use this stylesheet against the following document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
```

```
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

When we use an XSLT 1.0 processor with this document and stylesheet, the results are as follows:

```
Tests of the <decimal-format> element:

    format-number(1528.3, '#/###:00', 'f1')=1/528:30
    format-number(1 div 0, '###,###.00', 'f2')=Really, really big
    format-number(blue div orange, '#.##', 'f2')=[not a number]

**************************************
Sales report for 8/2006

Lindt:
    27,408 bars sold, 30% of all sales.

Callebaut:
    8,203 bars sold, 9% of all sales.

Valrhona:
    22,101 bars sold, 24% of all sales.

Perugina:
    14,336 bars sold, 16% of all sales.

Ghirardelli:
    19,268 bars sold, 21% of all sales.

Total sales: 91,316 bars.
```

[2.0] As we've seen numerous times, XSLT 2.0 is more strict in the things it will accept. A stylesheet with `1 div 0` will not run. If you need to use the XSLT 1.0 behavior in an XSLT 2.0 stylesheet, remember that you can use the `version` attribute on any XSLT 2.0 element:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- decimal-format2.xsl -->
```

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
...
    <xsl:text>&#xA;    format-number(1 div 0, '###,###.00', 'f2')=</xsl:text>
    <xsl:value-of version="1.0"
      select="format-number(1 div 0, '###,###.00', 'f2')"/>
...
</xsl:stylesheet>
```

Here we've changed the `version` attribute of the `<xsl:stylesheet>` element to `2.0`, but added `version="1.0"` to the `<xsl:value-of>` element later in the stylesheet. This gives us the XSLT 1.0 behavior for this one element, which means we'll get the value `Really, really big` when we divide by zero. If we use the default XSLT 2.0 behavior, the stylesheet throws an exception and no useful output is generated.

In XSLT 2.0, be aware that dividing an `xs:double` by zero returns `Infinity`, while dividing an `xs:integer` or `xs:decimal` by zero is a run-time error. If we change the stylesheet to read `xs:double(1) div xs:double(0)`, the stylesheet would run without errors, generating the same results (`Infinity`) as an XSLT 1.0 stylesheet that uses `1 div 0`.

See "[2.0] Attributes common to all XSLT elements" earlier in this appendix for more information about the `version` attribute and other attributes that can be added to any XSLT element.

[2.0] <xsl:document>

Allows you to create a new document node. This element is useful for validating the document node against a schema. The document node created by `<xsl:document>` is not meant to be serialized (written to disk); if that's what you want to do, use `<xsl:result-document>` instead.

Category

Instruction.

Required Attributes

None.

Optional Attributes

[2.0 – Schema] `type`

Defines the datatype of the root element node created by this element. To validate a document node, it must have as its children a single element node, no text nodes and zero or more comment and processing instruction nodes. If the document node doesn't have this structure, the XSLT processor throws an error.

The `type` and `validation` attributes are mutually exclusive.

[2.0 – Schema] `validation`

Defines how the value of the new element will be validated. The `validation` attribute has four values: `strict`, `lax`, `preserve`, or `strip`. To validate a document node with `strict` or `lax`, the document node must have as its children a single element node, no text nodes, and zero or more comment and processing instruction nodes. If the document node doesn't have this structure, the XSLT processor throws an error if you use `strict` or `lax`.

`validation="strict"` means that the XSLT processor looks in all the declared schemas for an element declaration (`<xs:element>`) with the same name as the single element node that is a child of this document node. It is a fatal error if the processor can't find a matching `<xs:element>`. Assuming the processor finds the declaration of the element node, it validates the generated value against the element's declaration.

`validation="lax"` works just like `validation="strict"`, except that no error occurs if the processor can't find the declaration of the element in any of the declared schemas. In that case, the type annotation of the element is `xs:untyped`.

The value `validation="preserve"` means the type annotations of the single element node and all its children and attributes will be preserved without changes. No schema validation is done.

Finally, `validation="strip"` sets the type annotation of the single element node to `xs:untyped`. All of the elements children and attributes have their type annotations set to `xs:untyped` for elements and to `xs:untypedAtomic` for attributes. No schema validation is done.

The `validation` and `type` attributes are mutually exclusive.

Content

A sequence constructor.

Appears in

`<xsl:document>` appears inside a template.

Defined in

[2.0] XSLT section 11.5, "Creating Document Nodes."

Example

We'll create a document node and give it some content. The node will be a `<name>` element, as defined in our purchase order schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- po.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```
  <xs:element name="purchase-order">
    ...
  </xs:element>

  ...

  <xs:element name="name">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="title"
          minOccurs="0" maxOccurs="1"/>
        <xs:element ref="first-name"
          minOccurs="1" maxOccurs="1"/>
        <xs:element ref="last-name"
          minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="title" type="xs:string"/>
  <xs:element name="first-name" type="xs:string"/>
  <xs:element name="last-name" type="xs:string"/>

</xs:schema>
```

Here is our stylesheet:

```
<?xml version="1.0"?>
<!-- document.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.oreilly.com/xslt"
  xmlns:po="http://www.oreilly.com/xslt">

  <xsl:output method="xml" indent="yes"/>

  <xsl:import-schema namespace="http://www.oreilly.com/xslt"
    schema-location="po.xsd" />

  <xsl:template match="/">
    <xsl:document validation="lax">
      <xsl:element name="name">
        <xsl:element name="title">
          <xsl:text>Mr.</xsl:text>
        </xsl:element>
        <xsl:element name="first-name">
          <xsl:text>Kent Lyle</xsl:text>
        </xsl:element>
        <xsl:element name="last-name">
          <xsl:text>Birdley</xsl:text>
        </xsl:element>
      </xsl:element>
    </xsl:document>
  </xsl:template>
```

```
</xsl:stylesheet>
```

We're using `validation="lax"` here. That means the XSLT processor will validate the document node that contains the `<name>` element if it can find an XML schema that defines the `<name>` element. If we take out the `<xsl:import-schema>` element, the `<xsl:document>` element still works. If we take out the `<xsl:import-schema>` element and change the `<xsl:document>` to `validation="strict"`, the stylesheet fails because it can't find a declaration for `<name>`.

Here are the results of the stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<name xmlns="http://www.oreilly.com/xslt">
   <title>Mr.</title>
   <first-name>Kent Lyle</first-name>
   <last-name>Birdley</last-name>
</name>
```

In this case, we've simply written the contents of the `<xsl:document>` element to the output. A more common use of `<xsl:document>` is to store the document node in a variable, and then use that validated variable elsewhere in the stylesheet.

Note that we can use the `<xsl:document>` element to create a document node that is not well-formed. Here's a sample stylesheet that does just that:

```
<?xml version="1.0"?>
<!-- document2.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:variable name="ill-formed" as="node()*">
    <xsl:document>
      <xsl:element name="title">
        <xsl:text>Mr.</xsl:text>
      </xsl:element>
      <xsl:element name="first-name">
        <xsl:text>Kent Lyle</xsl:text>
      </xsl:element>
      <xsl:element name="last-name">
        <xsl:text>Birdley</xsl:text>
      </xsl:element>
    </xsl:document>
  </xsl:variable>

  <xsl:template match="/">
    <xsl:text>&#xA;A document node that isn't well formed:</xsl:text>
    <xsl:text>&#xA;&#xA;  Is this a document node? </xsl:text>
    <xsl:value-of select="if ($ill-formed instance of document-node())
                          then 'Yes!'
                          else 'No!'"/>
    <xsl:text>&#xA;  Number of child elements: </xsl:text>
    <xsl:value-of select="count($ill-formed/*)"/>

    <xsl:result-document method="xml" href="ill-formed.xml">
```

```
      <xsl:copy-of select="$ill-formed"/>
    </xsl:result-document>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet creates a variable named `$ill-formed` that has three elements at the root of the document. We use the `document-node( )` node test to make sure this is, in fact, a document node, and then we print the number of children that the document node has:

```
A document node that isn't well formed:

  Is this a document node? Yes!
  Number of child elements: 3
```

Finally we use the `<xsl:result-document>` element to write our illegal markup to a file. The file `ill-formed.xml` looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<title>Mr.</title>
<first-name>Kent Lyle</first-name>
<last-name>Birdley</last-name>
```

Typically a document node has a single child that represents the document element; in this case, we've sidestepped the XML parser to create markup that isn't legal. Although this situation is uncommon, be aware that a stylesheet might generate a document node that has more than one child.

<xsl:element>

Allows you to create an element in the output document. It works similarly to the `<xsl:attribute>` element.

Category

Instruction.

Required Attribute

`name`
: Defines the name of this element. A value of `name="Fred"` produces a `<Fred>` element in the output document.

Optional Attributes

`namespace`
: Defines the namespace used for this element.

`use-attribute-sets`
: Lists one or more attribute sets that should be used by this element. If you specify more than one attribute set, separate their names with whitespace characters.

[2.0] `inherit-namespaces`
: Defines whether this element and its children inherit the current namespace nodes. Valid values are `yes` (the default) and `no`.

[2.0 – Schema] `type`
: Defines the datatype of this element. The datatype can be any of the built-in datatypes or it can be a datatype defined in a schema if you have a schema-aware XSLT 2.0 processor.

  The `type` and `validation` attributes are mutually exclusive.

[2.0 – Schema] `validation`
: Defines how the value of the new element will be validated. The `validation` attribute has four values: `strict`, `lax`, `preserve`, or `strip`.

  `validation="strict"` means that the XSLT processor looks in all the declared schemas for an element declaration (`<xs:element>`) with the same name as this element. If the processor can't find a matching `<xs:element>`, it is a fatal error. Assuming the processor finds the declaration of the element, it validates the generated element's value against its declaration.

  `validation="lax"` works just like `validation="strict"`, except that no error occurs if the processor can't find the declaration of the element in any of the declared schemas. In that case, the type annotation of the element is `xs:untyped`.

  The value `validation="preserve"` means the created element will have a type annotation of `xs:anyType`, and the type annotations of any nodes that the new element contains will be preserved without any changes. No schema validation is done.

  Finally, `validation="strip"` creates a new element of type `xs:anyType`, and any nodes the new element contains will have their type annotations replaced with `xs:untyped` for elements and with `xs:untypedAtomic` for attributes. No schema validation is done.

  The `validation` and `type` attributes are mutually exclusive.

Content

An XSLT template.

Appears in

`<xsl:element>` appears inside a template.

Defined in

[1.0] XSLT section 7.1.2, "Creating Elements with `xsl:element`."

[2.0] XSLT section 11.2, "Creating Element Nodes using `xsl:element`."

Example

We'll use a generic stylesheet that copies the input document to the result tree, with one exception: all attributes in the original documents are converted to child elements in the result

tree. The name of the new element will be the name of the attribute, and its text will be the value of the attribute. Because we don't know the name of the attribute until we process the XML source document, we must use the <xsl:element> element to create the result tree. Here's how our stylesheet looks:

```
<?xml version="1.0"?>
<!-- element.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml"/>

  <xsl:template match="*">
    <xsl:element name="{name()}">
      <xsl:for-each select="@*">
        <xsl:element name="{name()}">
          <xsl:value-of select="."/>
        </xsl:element>
      </xsl:for-each>
      <xsl:apply-templates select="*|text()"/>
    </xsl:element>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet uses the <xsl:element> element in two places: first to create a new element with the same name as the original element, and second to create a new element with the same name as each attribute. We'll apply the stylesheet to this document:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

Our results look like this:

```
<?xml version="1.0" encoding="UTF-8"?><list><xml:lang>en</xml:lang>
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem><xml:lang>sw</xml:lang>Aki Special</listitem>
  <listitem><xml:lang>en-gb</xml:lang>Combat Rock</listitem>
  <listitem><xml:lang>zu</xml:lang>Talking Timbuktu</listitem>
  <listitem><xml:lang>jz</xml:lang>The Birth of the Cool</listitem>
</list>
```

The <xsl:element> element created all the elements in the output document, including the <xml:lang> elements created from the xml:lang attributes.

[2.0 – Schema] Now we'll look at an example that uses XSLT 2.0's schema features. We'll use the validation attribute of <xsl:element> to make sure the element we create is valid according to our purchase order schema. We'll start with an XML file that contains similar data to our purchase order format:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- create-po.xml -->
<po order-num="38292">
  <customer id="4738" standing="Platinum">
    <address>
      <name>
        <courtesy>Mr.</courtesy>
        <given-name>Chester Hasbrouck</given-name>
        <surname>Frisby</surname>
      </name>
      <street>1234 Main Street</street>
      <city>Sheboygan</city>
      <state>WI</state>
      <zip>48392</zip>
    </address>
  </customer>
  <line-items>
    <line-item>
      <partnum>28392-33-TT</partnum>
      <partname>Turnip Twaddler</partname>
      <quantity>3</quantity>
      <price>9.95</price>
    </line-item>
    <line-item>
      <partnum>28100-38-CT</partnum>
      <partname>Clam Teaser</partname>
      <quantity>7</quantity>
      <price>39.95</price>
    </line-item>
  </line-items>
</po>
```

We'll use <xsl:element> to create a new purchase order using the data in this document. We also have to generate a <date> element as the first child of <purchase-order>. Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- element2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.oreilly.com/xslt"
  xmlns:po="http://www.oreilly.com/xslt"
  exclude-result-prefixes="xs po">

  <xsl:import-schema namespace="http://www.oreilly.com/xslt"
    schema-location="po.xsd" />
```

```
<xsl:output method="xml" indent="yes"/>

<xsl:variable name="now" as="xs:date" select="current-date()"/>

<xsl:template match="po">
  <xsl:element name="purchase-order" validation="strict">
    <xsl:attribute name="id" select="@order-num"/>
    <date>
      <xsl:attribute name="year" select="year-from-date($now)"/>
      <xsl:attribute name="month" select="month-from-date($now)"/>
      <xsl:attribute name="day" select="day-from-date($now)"/>
    </date>
    <customer>
      <xsl:attribute name="id" select="customer/@id"/>
      <xsl:attribute name="level" select="customer/@standing"/>
      <xsl:apply-templates select="customer/address"/>
    </customer>
    <xsl:apply-templates select="line-items"/>
  </xsl:element>
</xsl:template>

<xsl:template match="address">
  <address>
    <xsl:attribute name="type" select="'business'"/>
    <xsl:apply-templates select="name"/>
    <street>
      <xsl:value-of select="street"/>
    </street>
    <city>
      <xsl:value-of select="city"/>
    </city>
    <state>
      <xsl:value-of select="state"/>
    </state>
    <zip>
      <xsl:value-of select="zip"/>
    </zip>
  </address>
</xsl:template>

<xsl:template match="line-items">
  <items>
    <xsl:for-each select="line-item">
      <item>
        <xsl:attribute name="part-no" select="partnum"/>
        <partname>
          <xsl:value-of select="partname"/>
        </partname>
        <qty>
          <xsl:value-of select="quantity"/>
        </qty>
        <price>
          <xsl:value-of select="price"/>
        </price>
```

```
        </item>
      </xsl:for-each>
    </items>
  </xsl:template>

  <xsl:template match="name">
    <name>
      <xsl:if test="courtesy">
        <title>
          <xsl:value-of select="courtesy"/>
        </title>
      </xsl:if>
      <first-name>
        <xsl:value-of select="given-name"/>
      </first-name>
      <last-name>
        <xsl:value-of select="surname"/>
      </last-name>
    </name>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet uses `<xsl:element>` and `<xsl:attribute>` to build a purchase order from scratch. The `validation="strict"` attribute means the generated `<purchase-order>` element must be a valid purchase order when compared to the schema in *po.xsd*. If the stylesheet attempts to add anything to the generated `<purchase-order>` that violates the purchase order schema, the stylesheet engine throws an exception and stops. The stylesheet generates this XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<purchase-order xmlns="http://www.oreilly.com/xslt" id="38292">
   <date year="2008" month="3" day="2"/>
   <customer id="4738" level="Platinum">
      <address type="business">
         <name>
            <title>Mr.</title>
            <first-name>Chester Hasbrouck</first-name>
            <last-name>Frisby</last-name>
         </name>
         <street>1234 Main Street</street>
         <city>Sheboygan</city>
         <state>WI</state>
         <zip>48392</zip>
      </address>
   </customer>
   <items>
      <item part-no="28392-33-TT">
         <partname>Turnip Twaddler</partname>
         <qty>3</qty>
         <price>9.95</price>
      </item>
      <item part-no="28100-38-CT">
         <partname>Clam Teaser</partname>
         <qty>7</qty>
```

```
          <price>39.95</price>
        </item>
      </items>
    </purchase-order>
```

Notice that the generated document uses the default namespace of `http://www.oreilly.com/xslt`. This is the namespace associated with the imported schema; the fact that it is declared as the default namespace in the stylesheet means it is copied to the output element. Also notice that the stylesheet uses `exclude-result-prefixes` to keep the `po` and `xs` prefixes out of the generated document. The default namespace of the generated purchase order is `http://www.oreilly.com/xslt`, as it should be.

We create the `<date>` element with the `current-date( )` function. To see what happens when validation fails, take out the `<date>` element. You'll see an error message like this:

```
XTTE1510: In content of element <purchase-order>: The content model does
  not allow element <customer> to appear here. Expected:
  {http://www.oreilly.com/xslt}date (See
  http://www.w3.org/TR/xmlschema-1/#cvc-complex-type clause 2.4)
Transformation failed: Run-time errors were reported
```

In this case, the `<purchase-order>` element isn't generated because it isn't valid without the `<date>` element in the right place.

<xsl:fallback>

Defines a template that should be used when an XSLT processor finds an instruction in the stylesheet that it can't process. Fallback processing can be triggered by an extension element that relies on code that can't be found. It is also invoked when an XSLT 1.0 processor working in forward processing mode encounters an XSLT 2.0 element.

Category

Instruction.

Required Attributes

None.

Optional Attributes

None.

Content

An XSLT template.

Appears in

`<xsl:fallback>` appears inside a template.

Defined in

[1.0] XSLT section 15, "Fallback."

[2.0] XSLT section 18, "Extensibility and Fallback."

Example

Here is a stylesheet that uses `<xsl:fallback>` to terminate the transformation if an extension element can't be found:

```
<?xml version="1.0"?>
<!-- fallback.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:db="xalan://DatabaseExtension"
  extension-element-prefixes="db">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title><xsl:value-of select="report/title"/></title>
      </head>
      <body>
        <h1><xsl:value-of select="report/title"/></h1>
        <xsl:for-each select="report/section">
          <h2><xsl:value-of select="title"/></h2>
          <xsl:for-each select="dbaccess">
            <db:accessDatabase>
              <xsl:fallback>
                <p>Sorry, the database library is not available!</p>
              </xsl:fallback>
            </db:accessDatabase>
          </xsl:for-each>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

We'll use this stylesheet against this XML document:

```
<?xml version="1.0"?>
<!-- use-db.xml -->
<report>
  <title>HR employee listing</title>
  <section>
    <title>Employees by department</title>
    <dbaccess driver="COM.ibm.db2.jdbc.app.DB2Driver"
      database="jdbc:db2:sample" tablename="employee" where="*"
      fieldnames='workdept as "Department", lastname as "Last Name",
                  firstnme as "First Name"'
      order-by="workdept" group-by="workdept, lastname, firstnme"/>
  </section>
</report>
```

When we use this stylesheet to transform a document, the <xsl:fallback> element is processed if the extension element can't be found. If the element isn't available, you'll get these results:

```
<html>
  <head>
    <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>HR employee listing</title>
  </head>
  <body>
    <h1>HR employee listing</h1>
    <h2>Employees by department</h2>
    <p>Sorry, the database library is not available!</p>
  </body>
</html>
```

In this case, the extension element is <dbaccess>. This example was originally written for the Xalan processor; in Xalan, the value of an extension element prefix's URI is a Java class name. If, for whatever reason, the class DatabaseExtension can't be loaded, the <xsl:fallback> element is processed instead.

Note that the <xsl:fallback> element is processed only when the extension element can't be found; if the code that implements that extension element is found, but fails, that error must be handled some other way. Also be aware that the gracefulness of stylesheet termination will vary from one XSLT processor to the next.

<xsl:for-each>

XSLT's iteration operator. This element has a select attribute that selects some nodes from the current context. The contents of the <xsl:for-each> element are then evaluated using each of the selected nodes. *[2.0]* In XSLT 2.0, the select attribute selects items, which can include atomic values as well as nodes.)

Category

Instruction.

Required Attribute

`select`
: Contains an XPath expression that selects nodes from the current context.

Optional Attributes

None.

Content

<xsl:for-each> contains a template that is evaluated against each of the selected nodes. The <xsl:for-each> element can contain one or more <xsl:sort> elements to order the selected nodes before they are processed. All <xsl:sort> elements must appear first, before the template begins.

Appears in

<xsl:for-each> appears inside a template.

Defined in

[1.0] XSLT section 8, "Repetition."

[2.0] XSLT section 7, "Repetition."

Example

We'll demonstrate the <xsl:for-each> element with the following stylesheet:

```
<?xml version="1.0"?>
<!-- for-each.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Here is a moderately long list:
</xsl:text>
    <xsl:variable name="listitems" select="list/listitem"/>
    <xsl:call-template name="processListitems">
      <xsl:with-param name="items" select="$listitems"/>
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="processListitems">
    <xsl:param name="items"/>
    <xsl:for-each select="$items">
      <xsl:value-of select="position()"/>
      <xsl:text>.  </xsl:text>
      <xsl:value-of select="."/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

In this stylesheet, we use an <xsl:param> named items to illustrate the <xsl:for-each> element. The items parameter contains some number of <listitem> elements from the XML source document; the <xsl:for-each> element iterates through all those elements and processes each one. We'll use our stylesheet with the following XML document:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
```

```
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

When we run the transformation, here are the results:

```
Here is a moderately long list:
1.  The Sacred Art of Dub
2.  Only the Poor Man Feel It
3.  Excitable Boy
4.  Aki Special
5.  Combat Rock
6.  Talking Timbuktu
7.  The Birth of the Cool
```

The `<xsl:for-each>` element iterated through all the `<listitem>` elements from the XML source document and processed each one.

[2.0] <xsl:for-each-group>

Takes the items in a sequence and puts them into groups. There are four different ways of defining groups, each of which are explained in detail below. The grouping is done based on a common value or by a pattern that the first or last item in the group must match. Be aware that it is possible for an item in the original sequence to be put into more than one group.

Category

Instruction.

Required Attribute

`select`
: An XPath expression that determines the items to be grouped.

Optional Attributes

`group-by`
: Defines a common value that all items in a group must share. For example, `group-by="state"` creates a group for each unique value of the `<state>` element. All items in a given group will have the same value for `<state>`.

`group-adjacent`
: Defines an expression that is evaluated for every item in the sequence. If the value of that expression for the current item is the same as the value of that expression for the previous item, then the current item is put into the same group as the previous item. If not, the current item becomes the first item in a new group.

`group-starting-with`
: Defines a pattern that indicates the start of a new group. When an item matching that pattern is found, a new group is started, and all subsequent items are put into the same group until another item that matches the pattern is found.

`group-ending-with`
: Defines a pattern that indicates the end of the current group. When an item matching that pattern is found, the current group is closed. The next item in the sequence is put into a new group.

Keep in mind that `group-by`, `group-adjacent`, `group-starting-with`, and `group-ending-with` are mutually exclusive.

`collation`
: Defines the collation sequence used to compare grouping keys. The way collation sequences are defined can vary from one XSLT processor to another, so check your processor's documentation for the details.

  The `collation` attribute can only be used with `group-by` or `group-adjacent`. It is an error to use it with `group-starting-with` or `group-ending-with`.

Content

Zero or more `<xsl:sort>` elements and a sequence constructor.

Appears in

Any sequence constructor.

Defined in

XSLT section 14, "Grouping."

Example

We'll look at four examples here to illustrate the four types of grouping. We'll start with `group-by`, the most common type of grouping. We'll use a simplified list of customer addresses:

```
<?xml version="1.0"?>
<!-- simplified-names.xml -->
<addressbook>
  <address>
    <first-name>Chester Hasbrouck</first-name>
    <last-name>Frisby</last-name>
    <city>Sheboygan</city>
    <state>WI</state>
  </address>
  <address>
    <first-name>Mary</first-name>
    <last-name>Backstayge</last-name>
    <city>Skunk Haven</city>
    <state>MA</state>
  </address>
  <address>
    <first-name>Natalie</first-name>
    <last-name>Attired</last-name>
    <city>Winter Harbor</city>
    <state>ME</state>
```

```
  </address>
  <address>
    <first-name>Harry</first-name>
    <last-name>Backstayge</last-name>
    <city>Skunk Haven</city>
    <state>MA</state>
  </address>
  <address>
    <first-name>Mary</first-name>
    <last-name>McGoon</last-name>
    <city>Boylston</city>
    <state>VA</state>
  </address>
  <address>
    <first-name>Amanda</first-name>
    <last-name>Reckonwith</last-name>
    <city>Lynn</city>
    <state>MA</state>
  </address>
</addressbook>
```

We'll use `group-by` to group these addresses by state. Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- for-each-group1.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Customers grouped by state&#xA;&#xA;</xsl:text>
    <xsl:for-each-group select="/addressbook/address" group-by="state">
      <xsl:sort select="state"/>
      <xsl:text>  State = </xsl:text>
      <xsl:value-of select="current-grouping-key()"/>
      <xsl:text>&#xA;</xsl:text>
      <xsl:for-each select="current-group()">
        <xsl:text>&#x9;</xsl:text>
        <xsl:value-of select="(first-name, last-name)"
          separator=" "/>
        <xsl:text>, </xsl:text>
        <xsl:value-of select="city"/>
        <xsl:text>&#xA;</xsl:text>
      </xsl:for-each>
    </xsl:for-each-group>
  </xsl:template>

</xsl:stylesheet>
```

Our stylesheet generates these results:

```
Customers grouped by state

  State = MA
        Mary Backstayge, Skunk Haven
        Harry Backstayge, Skunk Haven
```

```
      Amanda Reckonwith, Lynn
  State = ME
      Natalie Attired, Winter Harbor
  State = VA
      Mary McGoon, Boylston
  State = WI
      Chester Hasbrouck Frisby, Sheboygan
```

The <xsl:sort> element after <xsl:for-each-group> sorts the groups themselves. For each group a heading is printed, and then the contents of each group are handled by <xsl:for-each select="current-group( )">. If we take out the <xsl:sort> element to order the groups, our output is different. Our customers are still grouped correctly, but the groups are in a different order:

```
Customers grouped by state

  State = WI
      Chester Hasbrouck Frisby, Sheboygan
  State = MA
      Mary Backstayge, Skunk Haven
      Harry Backstayge, Skunk Haven
      Amanda Reckonwith, Lynn
  State = ME
      Natalie Attired, Winter Harbor
  State = VA
      Mary McGoon, Boylston
```

Because we didn't sort the groups, they appear in the order in which each state first appeared.

For group-adjacent, we define an expression that returns a value for each item in the sequence. To keep our example simple, we'll use group-adjacent to return true( ) or false( ). Within the sequence, all adjacent items that have the same value are put into the same group. We're looking for groups of adjacent <p> elements in an HTML document; we'll convert them to <ul> elements in which each <p> element is transformed into a list item. Here's the HTML source:

```
<?xml version="1.0"?>
<!-- grouping-input.html -->
<html>
  <body>
    <!-- Here's some sample text from the chapter "Sorting
         and Grouping". -->
    <h1>Steps for grouping in the Muench method</h1>
    <p>Define a <code>key</code> for the property we want
    to use for grouping.</p>
    <p>Select all of the nodes ...</p>
    <p>For each unique grouping value, ...</p>
    <h1>Steps for grouping in XSLT 2.0</h1>
    <p>Define an XPath expression ...</p>
    <p>Select all of the nodes we want to group ...</p>
    <p>Instead of dealing with each ...</p>
  </body>
</html>
```

We'll use this stylesheet to do the grouping:

```
<?xml version="1.0"?>
<!-- for-each-group2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" include-content-type="no"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Grouping with group-adjacent</title>
      </head>
      <body>
        <xsl:for-each-group select="html/body/*"
          group-adjacent="boolean(self::p)">
          <xsl:choose>
            <xsl:when test="current-grouping-key()">
              <ul>
                <xsl:for-each select="current-group()">
                  <li><xsl:apply-templates select="*|text()"/></li>
                </xsl:for-each>
              </ul>
            </xsl:when>
            <xsl:otherwise>
              <xsl:apply-templates select="current-group()" />
            </xsl:otherwise>
          </xsl:choose>
        </xsl:for-each-group>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="*">
    <xsl:copy>
      <xsl:for-each select="@*">
        <xsl:copy/>
      </xsl:for-each>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

Our stylesheet yields these results:

```
<html>
   <head>
      <title>Grouping with group-adjacent</title>
   </head>
   <body>
      <h1>Steps for grouping in the Muench method</h1>
      <ul>
         <li>Define a <code>key</code> for the property we want
            to use for grouping.
         </li>
         <li>Select all of the nodes ...</li>
```

```
            <li>For each unique grouping value, ...</li>
         </ul>
         <h1>Steps for grouping in XSLT 2.0</h1>
         <ul>
            <li>Define an XPath expression ...</li>
            <li>Select all of the nodes we want to group ...</li>
            <li>Instead of dealing with each ...</li>
         </ul>
      </body>
   </html>
```

The `group-adjacent` attribute is an XPath expression that returns a value. In this case, any `<p>` element returns `true( )`, while everything else returns `false( )`. The value of `group-adjacent` can be a much more complicated expression. See the discussion of "Another Type of Grouping: group-adjacent" in Chapter 6 for more detail.

Our third grouping example uses `group-starting-with`. In this example, we'll use each `<h1>` element as the start of a section. Each `<h1>` element will start a group; everything up until the next `<h1>` element will be in the same group. We'll create a DocBook output document in which the `<h1>`, and everything that follows it is in a `<sect1>` element. (We'll reuse our HTML document from the `group-adjacent` example.) Here's our `group-starting-with` stylesheet:

```
<?xml version="1.0"?>
<!-- for-each-group3.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/">
    <chapter>
      <title>Grouping in XSLT</title>
      <xsl:apply-templates select="html/body"/>
    </chapter>
  </xsl:template>

  <xsl:template match="body">
    <xsl:for-each-group select="*" group-starting-with="h1">
      <sect1>
        <xsl:apply-templates select="current-group()"/>
      </sect1>
    </xsl:for-each-group>
  </xsl:template>

  <xsl:template match="h1">
    <title>
      <xsl:apply-templates/>
    </title>
  </xsl:template>

  <xsl:template match="p">
    <para>
      <xsl:apply-templates/>
    </para>
  </xsl:template>
```

```
  <xsl:template match="*">
    <xsl:copy>
      <xsl:for-each select="@*">
        <xsl:copy/>
      </xsl:for-each>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

Our stylesheet generates a DocBook document in which each `<h1>` and everything following it is now a `<sect1>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<chapter>
   <title>Grouping in XSLT</title>
   <sect1>
      <title>Steps for grouping in the Muench method</title>
      <para>Define a <code>key</code> for the property we want
    to use for grouping.</para>
      <para>Select all of the nodes ...</para>
      <para>For each unique grouping value, ...</para>
   </sect1>
   <sect1>
      <title>Steps for grouping in XSLT 2.0</title>
      <para>Define an XPath expression ...</para>
      <para>Select all of the nodes we want to group ...</para>
      <para>Instead of dealing with each ...</para>
   </sect1>
</chapter>
```

The text of the HTML `<h1>` becomes a DocBook `<title>` element, and each HTML `<p>` element becomes a DocBook `<para>` element.

Our final example is for `group-ending-with`. In this example, we'll take a list of elements and put them into three columns. The ending condition is each third element (`position( ) mod 3 = 0`). We'll use this list of cars:

```
<?xml version="1.0"?>
<!-- carlist.xml -->
<cars>
  <make>Alfa Romeo</make>
  <make>Bentley</make>
  <make>Chevrolet</make>
  <make>Dodge</make>
  <make>Eagle</make>
  <make>Ford</make>
  <make>GMC</make>
  <make>Honda</make>
  <make>Isuzu</make>
  <model>Javelin</model>
  <model>K-Car</model>
  <make>Lincoln</make>
  <make>Mercedes</make>
```

```
  <make>Nash</make>
  <make>Opel</make>
  <make>Pontiac</make>
  <model>Quantum</model>
  <model>Rambler</model>
  <make>Studebaker</make>
</cars>
```

Here's our stylesheet:

```
<?xml version="1.0"?>
<!-- for-each-group4.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Car Makes and Models</title>
      </head>
      <body style="font-family: sans-serif;">
        <h1>Car Makes and Models</h1>
        <p>Here are the car makes and models in
        our input document.</p>
        <table border="1" cellpadding="5">
          <xsl:apply-templates select="cars"/>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="cars">
    <xsl:for-each-group select="make|model"
      group-ending-with="*[position() mod 3 = 0]">
      <tr>
        <xsl:choose>
          <xsl:when test="count(current-group()) = 3">
            <xsl:for-each select="current-group()">
              <xsl:apply-templates select="."/>
            </xsl:for-each>
          </xsl:when>
          <xsl:when test="count(current-group()) = 2">
            <xsl:apply-templates select="current-group()[1]"/>
            <xsl:apply-templates select="current-group()[2]"/>
            <td bgcolor="#CCCCCC">
              &#x20;&#x20;
            </td>
          </xsl:when>
          <xsl:otherwise>
            <xsl:apply-templates select="current-group()[1]"/>
            <td bgcolor="#CCCCCC" colspan="2">
              &#x20;&#x20;
```

```
            </td>
          </xsl:otherwise>
        </xsl:choose>
      </tr>
    </xsl:for-each-group>
  </xsl:template>

  <xsl:template match="make">
    <td style="font-weight: bold;">
      <xsl:apply-templates/>
    </td>
  </xsl:template>

  <xsl:template match="model">
    <td style="font-style: italic; font-weight: bold;">
      <xsl:apply-templates/>
    </td>
  </xsl:template>

</xsl:stylesheet>
```

Our results look like this:

```
<html>
   <head>
      <title>Car Makes and Models</title>
   </head>
   <body style="font-family: sans-serif;">
      <h1>Car Makes and Models</h1>
      <p>Here are the car makes and models in
         our input document.
      </p>
      <table border="1" cellpadding="5">
         <tr>
            <td style="font-weight: bold;">Alfa Romeo</td>
            <td style="font-weight: bold;">Bentley</td>
            <td style="font-weight: bold;">Chevrolet</td>
         </tr>
         ...
         <tr>
            <td style="font-weight: bold;">Studebaker</td>
            <td bgcolor="#CCCCCC" colspan="2"></td>
         </tr>
      </table>
   </body>
</html>
```

The elements in the list of cars have been written to our HTML table in groups of three. Notice that the last group has only one element, so that row of the table features a cell with the last element followed by a blank, gray cell with `colspan="2"`. The document appears as shown in Figure A-9.

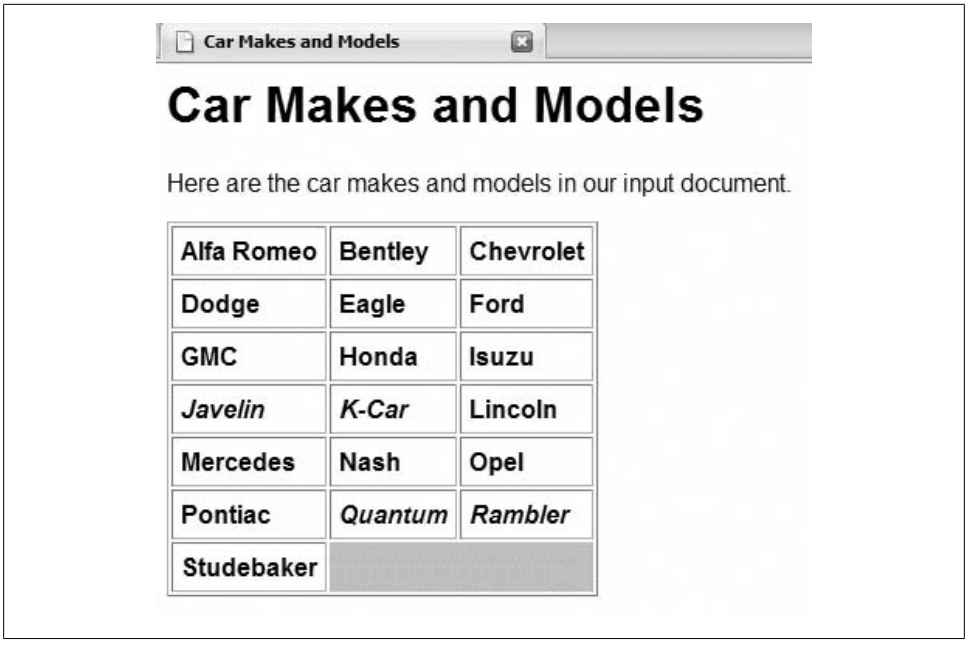


Figure A-9. Items grouped with group-ending-with

[2.0] <xsl:function>

Defines a function that can be used in XPath expressions in the stylesheet.

Category

Declaration.

Required Attribute

`name`
: The name of the stylesheet function.

Optional Attributes

`as`
: Defines the datatype returned by this function.

`override`
: Defines whether this function should override another function with the same name and the same number of arguments that is provided by the processor outside the scope of the stylesheet. `override="yes"`, the default value, means this function will be used (it overrides the other function); `override="no"` means the other function will be used.

Content

Any number of <xsl:param> elements, followed by a sequence constructor. The created sequence is returned by the function.

Defined in

XSLT section 10.3, "Stylesheet Functions."

Example

This stylesheet uses three <xsl:function>s to return the background color, font size, and style of various HTML elements. Putting the logic into a function means we can access the function from an attribute value template. Here's the stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- function.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:sample="http://www.oreilly.com/catalog/xslt"
  exclude-result-prefixes="xs sample">

  <xsl:output method="html" include-content-type="no"/>

  <xsl:variable name="colors" as="xs:string *"
    select="('yellow', 'white', 'blue')"/>

  <xsl:variable name="fontSizes" as="xs:integer *"
    select="(18, 24, 36)"/>

  <xsl:variable name="styles" as="xs:string *"
    select="('color: black;', 'color: black;',
            'color: white; font-weight: bold;')"/>

  <xsl:template match="/">
    <html>
      <head>
        <title><xsl:value-of select="/list/title"/></title>
      </head>
      <body style="font-family: sans-serif;">
        <h1 style="font-size: 48;"><xsl:value-of select="/list/title"/></h1>
        <table border="3" cellpadding="5" cellspacing="5" width="50%">
          <tr>
            <xsl:for-each select="/list/listitem">
              <td style="font-size: {sample:getFontSize(position())};
                         {sample:getStyle(position())}"
                bgcolor="{sample:getColor(position())}">
                <xsl:value-of select="."/>
              </td>
            </xsl:for-each>
          </tr>
        </table>
      </body>
    </html>
  </xsl:template>
```

```
  <xsl:function name="sample:getColor" as="xs:string">
    <xsl:param name="pos" as="xs:integer"/>
    <xsl:value-of select="$colors[($pos mod count($colors)) + 1]"/>
  </xsl:function>

  <xsl:function name="sample:getStyle" as="xs:string">
    <xsl:param name="pos" as="xs:integer"/>
    <xsl:value-of select="$styles[($pos mod count($styles)) + 1]"/>
  </xsl:function>

  <xsl:function name="sample:getFontSize" as="xs:integer">
    <xsl:param name="pos" as="xs:integer"/>
    <xsl:value-of select="$fontSizes[($pos mod count($fontSizes)) + 1]"/>
  </xsl:function>

</xsl:stylesheet>
```

Our functions work with the variables `colors` and `fontSizes`. These are defined as sequences of strings and integers, respectively. We call the `getColor( )` and `getFontSize( )` functions with the position of the current node; the functions use the `mod` operator to determine the item in the sequence that should be returned.

Notice that we add `1` to the index of the item returned by our functions. *The first item in a sequence is at position one, not zero.* Also, to make the code easier to maintain, we use the `count( )` function to determine the second operand for the `mod` operator. If we add more colors or font sizes to our sequences, we won't have to change our function definitions to match the new size.

We'll use this stylesheet to transform our list of albums:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

The generated HTML document looks like this:

```
<html>
   <head>
      <title>Albums I've bought recently:</title>
   </head>
   <body style="font-family: sans-serif;">
      <h1 style="font-size: 48;">Albums I've bought recently:</h1>
      <table border="3" cellpadding="5" cellspacing="5" width="50%">
         <tr>
            <td style="font-size: 24;
              color: black;" bgcolor="white">The Sacred Art of Dub</td>
```

Figure A-10. An HTML table generated with the help of XSLT functions

```
                <td style="font-size: 36;
                  color: white; font-weight: bold;" bgcolor="blue">
                  Only the Poor Man Feel It</td>
                <td style="font-size: 18;
                  color: black;" bgcolor="yellow">Excitable Boy</td>
                <td style="font-size: 24;
                  color: black;" bgcolor="white">Aki Special</td>
                <td style="font-size: 36;
                  color: white; font-weight: bold;" bgcolor="blue">Combat Rock</td>
                <td style="font-size: 18;
                  color: black;" bgcolor="yellow">Talking Timbuktu</td>
                <td style="font-size: 24;
                  color: black;" bgcolor="white">The Birth of the Cool</td>
              </tr>
            </table>
          </body>
        </html>
```

The values for the `bgcolor` and `style` attributes are generated by the XSLT functions we defined. When viewed in a browser, the output document appears as shown in Figure A-10.

XSLT functions can make your stylesheets much simpler and easier to maintain. In our XSLT 2.0 stylesheet, we used this markup to create the `bgcolor` attribute:

```
bgcolor="{sample:getColor(position())}"
```

Here's how you do the same thing in XSLT 1.0:

```
<xsl:attribute name="bgcolor">
  <xsl:choose>
    <xsl:when test="position() mod 3 = 0">
      <xsl:text>yellow</xsl:text>
    </xsl:when>
    <xsl:when test="position() mod 3 = 1">
      <xsl:text>white</xsl:text>
    </xsl:when>
```

```
      <xsl:otherwise>
        <xsl:text>blue</xsl:text>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:attribute>
```

<xsl:if>

Implements an `if` statement. It contains a `test` attribute and an XSLT template. If the `test` attribute evaluates to the boolean value `true`, the XSLT template is processed. This element implements an `if` statement only; if you need an if-then-else statement, use the `<xsl:choose>` element with a single `<xsl:when>` and a single `<xsl:otherwise>`.

Category

Instruction.

Required Attribute

`test`

Contains a boolean expression. If it evaluates to the boolean value `true`, then the XSLT template inside the `<xsl:if>` element is processed.

Optional Attributes

None.

Content

An XSLT template.

Appears in

`<xsl:if>` appears inside a template.

Defined in

[1.0] XSLT section 9.1, "Conditional Processing with `xsl:if`."

[2.0] XSLT section 8.1, "Conditional Processing with `xsl:if`."

Example

We'll illustrate the `<xsl:if>` element with the following stylesheet:

```
<?xml version="1.0"?>
<!-- if.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;</xsl:text>
```

```
    <xsl:text>Here are the odd-numbered items from the list:</xsl:text>
    <xsl:text>&#xA;</xsl:text>
    <xsl:for-each select="list/listitem">
      <xsl:if test="(position() mod 2) = 1">
        <xsl:number format="1. "/>
        <xsl:value-of select="."/>
        <xsl:text>&#xA;</xsl:text>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet uses the `<xsl:if>` element to see whether a given `<listitem>`'s position is an odd number. If it is, we write it to the result tree. We'll test our stylesheet with this XML document:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list>
  <title>A few of my favorite albums</title>
  <listitem>A Love Supreme</listitem>
  <listitem>Beat Crazy</listitem>
  <listitem>You Could Have it So Much Better</listitem>
  <listitem>Kind of Blue</listitem>
  <listitem>London Calling</listitem>
  <listitem>Remain in Light</listitem>
  <listitem>The Joshua Tree</listitem>
  <listitem>The Indestructible Beat of Soweto</listitem>
</list>
```

When we run this transformation, here are the results:

```
Here are the odd-numbered items from the list:
1. A Love Supreme
3. You Could Have it So Much Better
5. London Calling
7. The Joshua Tree
```

<xsl:import>

Allows you to import the templates found in another XSLT stylesheet. Unlike `<xsl:include>`, all templates imported with `<xsl:import>` have a lower priority than those in the including stylesheet. Another difference between `<xsl:include>` and `<xsl:import>` is that `<xsl:include>` can appear anywhere in a stylesheet, whereas `<xsl:import>` can appear only at the beginning.

Category

Top-level element.

Required Attribute

`href`
: Defines the URI of the imported stylesheet.

Optional Attributes

None.

Content

None. `<xsl:import>` is an empty element.

Appears in

`<xsl:import>` is a top-level element and can appear only as a child of `<xsl:stylesheet>`.

Defined in

[1.0] XSLT section 2.6.2, "Stylesheet Import."

[2.0] XSLT section 3.10.3, "Stylesheet Import."

Example

Here is a simple stylesheet that we'll import:

```
<?xml version="1.0"?>
<!-- listitem.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;</xsl:text>
    <xsl:apply-templates select="list/title"/>
    <xsl:apply-templates select="list/listitem"/>
  </xsl:template>

  <xsl:template match="title">
    <xsl:value-of select="."/>
    <xsl:text>: </xsl:text>
    <xsl:text>&#xA;</xsl:text>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

  <xsl:template match="listitem">
    <xsl:text>HERE IS LISTITEM NUMBER </xsl:text>
    <xsl:value-of select="position()"/>
    <xsl:text>:  </xsl:text>
    <xsl:value-of select="."/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

To illustrate how <xsl:import> works, we'll test this stylesheet with this document:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list>
  <title>A few of my favorite albums</title>
  <listitem>A Love Supreme</listitem>
  <listitem>Beat Crazy</listitem>
  <listitem>You Could Have it So Much Better</listitem>
  <listitem>Kind of Blue</listitem>
  <listitem>London Calling</listitem>
  <listitem>Remain in Light</listitem>
  <listitem>The Joshua Tree</listitem>
  <listitem>The Indestructible Beat of Soweto</listitem>
</list>
```

When we process our XML source document with this stylesheet, here are the results:

```
A few of my favorite albums:

HERE IS LISTITEM NUMBER 1:  A Love Supreme
HERE IS LISTITEM NUMBER 2:  Beat Crazy
HERE IS LISTITEM NUMBER 3:  You Could Have it So Much Better
HERE IS LISTITEM NUMBER 4:  Kind of Blue
HERE IS LISTITEM NUMBER 5:  London Calling
HERE IS LISTITEM NUMBER 6:  Remain in Light
HERE IS LISTITEM NUMBER 7:  The Joshua Tree
HERE IS LISTITEM NUMBER 8:  The Indestructible Beat of Soweto
```

Now we'll use <xsl:import> to import our first stylesheet into a second one:

```
<?xml version="1.0"?>
<!-- import.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:import href="listitem.xsl"/>

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;</xsl:text>
    <xsl:apply-templates select="list/title"/>
    <xsl:apply-templates select="list/listitem"/>
  </xsl:template>

  <xsl:template match="listitem">
    <xsl:value-of select="position()"/>
    <xsl:text>.  </xsl:text>
    <xsl:value-of select="."/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results created by our second stylesheet:

```
A few of my favorite albums:

1.  A Love Supreme
2.  Beat Crazy
3.  You Could Have it So Much Better
4.  Kind of Blue
5.  London Calling
6.  Remain in Light
7.  The Joshua Tree
8.  The Indestructible Beat of Soweto
```

Notice that both stylesheets had a template with `match="listitem"`. The template in the imported stylesheet has a lower priority, so it is not used. Only the imported stylesheet has a template with `match="title"`, so the imported template is used for the `<title>` element.

See Also

The descriptions of the `<xsl:include>`, `<xsl:apply-imports>`, and [2.0] `<xsl:next-match>` elements.

[2.0 – Schema] <xsl:import-schema>

Allows you to import an XML Schema. The schema is imported and processed before any input documents are processed. This allows you to define datatypes and validation rules before the XSLT processor begins to transform the input document. *This element is only supported by schema-aware XSLT 2.0 processors.*

Category

Top-level element.

Required Attributes

None.

Optional Attributes

`namespace`
: The namespace used by the imported schema. Any templates or XPath functions can use this namespace to indicate that an element or value is defined by that schema.

`schema-location`
: A URI reference to the schema document. If a stylesheet imports *and uses* a schema, it is a fatal error if the schema can't be found. An XSLT processor is allowed to ignore the error if you import a schema but never use it.

  The `<xsl:import-schema>` element can include the `<xs:schema>` element itself. It is a fatal error if an `<xsl:import-schema>` element contains a `<xs:schema>` and has a `schema-location` attribute.

Content

None of the elements from the XSLT namespace can appear inside <xsl:import-schema>. If <xsl:import-schema> has any contents, it will likely be an <xs:schema> element that in turn defines datatypes and elements. Again, it is an error for an <xsl:import-schema> element to contain content and have a schema-location attribute.

Appears in

<xsl:import-schema> is a top-level element and can only appear as a child of <xsl:stylesheet>.

Defined in

[2.0] XSLT section 3.14, "Importing Schema Components."

Example

We'll illustrate <xsl:import-schema> with the purchase order schema that we've used several times throughout the book. Here's a valid purchase order:

```
<?xml version="1.0"?>
<!-- good-po.xml -->
<purchase-order id="38292"
  xmlns="http://www.oreilly.com/xslt">
  <date year="2001" month="6" day="19"/>
  <customer id="4738" level="Platinum">
    <address type="business">
      <name>
        <title>Mr.</title>
        <first-name>Chester Hasbrouck</first-name>
        <last-name>Frisby</last-name>
      </name>
      <street>1234 Main Street</street>
      <city>Sheboygan</city>
      <state>WI</state>
      <zip>48392</zip>
    </address>
  </customer>
  <items>
    <item part-no="28392-33-TT">
      <partname>Turnip Twaddler</partname>
      <qty>3</qty>
      <price>9.95</price>
    </item>
    <item part-no="28100-38-CT">
      <partname>Clam Teaser</partname>
      <qty>7</qty>
      <price>39.95</price>
    </item>
  </items>
</purchase-order>
```

Now we'll use <xsl:import-schema> to read the schema file and the XML document, and then see whether the XML document is an instance of the <po:purchase-order> element defined in the schema:

```
<?xml version="1.0"?>
<!-- import-schema.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:po="http://www.oreilly.com/xslt">

  <xsl:import-schema namespace="http://www.oreilly.com/xslt"
    schema-location="po.xsd" />

  <xsl:output method="text"/>

  <xsl:template match="schema-element(po:purchase-order)">
    <xsl:text>&#xA;This is a test of the &lt;xsl:import-</xsl:text>
    <xsl:text>schema&gt; element.&#xA;&#xA;</xsl:text>
    <xsl:text>Here are all the items in this purchase </xsl:text>
    <xsl:text>order:&#xA;</xsl:text>
    <xsl:for-each select="po:items/po:item">
      <xsl:text>  * </xsl:text>
      <xsl:value-of select="po:partname"/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Notice that in addition to importing the schema, we use the schema-element node test in the match attribute. That means the template only matches validated <po:purchase-order> elements. Here are the results:

```
This is a test of the <xsl:import-schema> element.

Here are all the items in this purchase order:
  * Turnip Twaddler
  * Clam Teaser
```

In the schema itself, the targetNamespace attribute and the default namespace (xmlns=) match the namespace attribute of the <xsl:import-schema> element in the stylesheet. In order to refer to the elements in the purchase order, we associate the prefix po with that namespace. Whenever we want to refer to elements in the purchase order namespace, we use the prefix (po:purchase-order, for example). If the namespace URIs don't match, the stylesheet won't work.

Here's an alternate version of the stylesheet, in which the <xsl:import-schema> element contains the schema itself. It generates the same results as the previous version:

```
<?xml version="1.0"?>
<!-- import-schema2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:po="http://www.oreilly.com/xslt"
```

```
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xsl:import-schema namespace="http://www.oreilly.com/xslt">
      <xs:schema
        targetNamespace="http://www.oreilly.com/xslt"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">

        <xs:element name="purchase-order">
          <xs:complexType>
            <xs:sequence>
              <xs:element ref="date"
                minOccurs="1" maxOccurs="1"/>
              <xs:element ref="customer"
                minOccurs="1" maxOccurs="1"/>
              <xs:element ref="items"
                minOccurs="1" maxOccurs="1"/>
            </xs:sequence>
            <xs:attribute name="id" type="xs:string"/>
          </xs:complexType>
        </xs:element>

        ...

      </xs:schema>
    </xsl:import-schema>

    <xsl:output method="text"/>

    <xsl:template match="schema-element(po:purchase-order)">
      <xsl:text>&#xA;This is a test of the <xsl:import-</xsl:text>
      <xsl:text>schema> element.&#xA;&#xA;</xsl:text>
      <xsl:text>Here are all the items in this purchase </xsl:text>
      <xsl:text>order:&#xA;</xsl:text>
      <xsl:for-each select="po:items/po:item">
        <xsl:text>  * </xsl:text>
        <xsl:value-of select="po:partname"/>
        <xsl:text>&#xA;</xsl:text>
      </xsl:for-each>
    </xsl:template>

  </xsl:stylesheet>
```

<xsl:include>

Allows you to include another XSLT stylesheet. This element allows you to put common transformations in a separate stylesheet, then include the templates from that stylesheet at any time. Unlike `<xsl:import>`, all templates included with `<xsl:include>` have the same priority as those in the including stylesheet. Another difference is that `<xsl:include>` can appear anywhere in a stylesheet, whereas `<xsl:import>` must appear at the beginning.

Category

Top-level element.

Required Attribute

`href`
: The URI of the included stylesheet.

Optional Attributes

None.

Content

None. `<xsl:include>` is an empty element.

Appears in

`<xsl:include>` is a top-level element and can appear only as a child of `<xsl:stylesheet>`.

Defined in

[1.0] XSLT section 2.6.1, "Stylesheet Inclusion."

[2.0] XSLT section 3.10.2, "Stylesheet Inclusion."

Example

The `<xsl:include>` element is a good way to break your stylesheets into smaller pieces; those smaller pieces are often easier to reuse. Here's a short stylesheet that includes another:

```
<?xml version="1.0"?>
<!-- include.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:include href="include-stylesheet.xsl"/>

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:value-of select="/report/title"/>
    <xsl:text>&#xA;&#xA;</xsl:text>
    <xsl:for-each select="report/brand">
      <xsl:text>  </xsl:text>
      <xsl:value-of select="name"/>
      <xsl:text>:  </xsl:text>
      <xsl:value-of select="format-number(units, '##,###')"/>
      <xsl:text> bars sold. </xsl:text>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
    <xsl:text>&#xA;Total sales: </xsl:text>
    <xsl:value-of
      select="format-number(sum(/report/brand/units), '##,###')"/>
    <xsl:text> bars.&#xA;&#xA;</xsl:text>
    <xsl:value-of select="$copyright"/>
  </xsl:template>

</xsl:stylesheet>
```

And here is the stylesheet that it includes:

```
<?xml version="1.0"?>
<!-- included-stylesheet.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:variable name="copyright">
    <xsl:text>(C) Copyright 2007 DougCo Incorporated.</xsl:text>
  </xsl:variable>

</xsl:stylesheet>
```

The included stylesheet defines a variable, `$copyright`, that has some commonly used text. We might have dozens of stylesheets that have a copyright notice; we can use this included stylesheet to make sure all of our generated documents include this notice. When we need to change the notice (on the first of January, example), we change it in one place. This technique is also very useful for defining sets of CSS properties that can be replaced or swapped out with very little effort.

Here are the results of our stylesheet:

```
Chocolate bar sales

  Lindt - 27,408 bars sold.
  Callebaut - 8,203 bars sold.
  Valrhona - 22,101 bars sold.
  Perugina - 14,336 bars sold.
  Ghirardelli - 19,268 bars sold.

Total sales: 91,316 bars.

(C) Copyright 2007 DougCo Incorporated.
```

The variable `$copyright` contains the text as defined in the included stylesheet.

<xsl:key>

Defines an index against the current document. The element is defined with three attributes: a `name`, which names this index; a `match`, an XPath expression that describes the nodes to be indexed; and a `use` attribute, an XPath expression that defines the property used to create the index. Much like database indexes can improve the performance of a database application, keys can improve the performance of a stylesheet.

Category

Top-level element.

Required Attributes

`name`
: Defines a name for this key.

`match`
: Represents an XPath expression that defines the nodes to be indexed by this key.

`use`
: Represents an XPath expression that defines the property of the indexed nodes that will be used to retrieve nodes from the index.

  [2.0] In XSLT 2.0, this attribute is optional. The `<xsl:key>` element can contain a sequence constructor that creates or selects the nodes to be indexed.

Optional Attributes

[1.0] None
: In XSLT 1.0, `<xsl:key>` doesn't have any optional attributes.

[2.0] `collation`
: The `collation` attribute defines the collation sequence used to determine whether two key values are equal.

Content

[1.0] None
: In XSLT 1.0, the `<xsl:key>` element is empty.

[2.0] A sequence constructor
: In XSLT 2.0, the `<xsl:key>` element can contain any sequence constructor. It is an error if a `<xsl:key>` element has both content and a `use` attribute.

Appears in

`<xsl:key>` is a top-level element and can only appear as a child of `<xsl:stylesheet>`.

Defined in

[1.0] XSLT section 12.2, "Keys."

[2.0] XSLT section 16.3, "Keys."

Example

For an example, we'll use our simplified list of customers:

```
<?xml version="1.0"?>
<!-- simplified-names.xml -->
<addressbook>
  <address>
    <first-name>Chester Hasbrouck</first-name>
    <last-name>Frisby</last-name>
    <city>Sheboygan</city>
    <state>WI</state>
  </address>
  <address>
    <first-name>Mary</first-name>
    <last-name>Backstayge</last-name>
    <city>Skunk Haven</city>
```

```
    <state>MA</state>
  </address>
  <address>
    <first-name>Natalie</first-name>
    <last-name>Attired</last-name>
    <city>Winter Harbor</city>
    <state>ME</state>
  </address>
  <address>
    <first-name>Harry</first-name>
    <last-name>Backstayge</last-name>
    <city>Skunk Haven</city>
    <state>MA</state>
  </address>
  <address>
    <first-name>Mary</first-name>
    <last-name>McGoon</last-name>
    <city>Boylston</city>
    <state>VA</state>
  </address>
  <address>
    <first-name>Amanda</first-name>
    <last-name>Reckonwith</last-name>
    <city>Lynn</city>
    <state>MA</state>
  </address>
</addressbook>
```

Here is a stylesheet that defines a `<xsl:key>` against our list of customers:

```
<?xml version="1.0"?>
<!-- key.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:param name="searchState"/>

  <xsl:key name="customersByState" match="address" use="state"/>

  <xsl:template match="/">
    <xsl:text>All customers in </xsl:text>
    <xsl:value-of select="$searchState"/>
    <xsl:text>&#xA;&#xA;</xsl:text>
    <xsl:for-each select="key('customersByState', $searchState)">
      <xsl:value-of select="first-name"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="last-name"/>
      <xsl:text>, </xsl:text>
      <xsl:value-of select="city"/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

We define our key with a name of `customersByState`, we use the key to index all of the `<address>` elements in our document, and we use the value of the `<state>` element as the index property. Here's how our stylesheet responds when we invoke the XSLT processor with the input parameter `state=MA`:

```
All customers in MA

Mary Backstayge, Skunk Haven
Harry Backstayge, Skunk Haven
Amanda Reckonwith, Lynn
```

Asking for all the customers in Wisconsin (`state=WI`) uses our `<xsl:key>` to retrieve different values:

```
All customers in WI

Chester Hasbrouck Frisby, Sheboygan
```

Finally, it's not an error if the key doesn't find any values. Here's a list of all our customers in Hawaii:

```
All customers in HI
```

[2.0] <xsl:matching-substring>

Defines what to do when a string matches a regular expression. The regular expression is defined on the `regex` attribute of the `<xsl:analyze-string>` element that contains the `<xsl:matching-substring>` element.

Category

Instruction (this is effectively part of the `<xsl:analyze-string>` element).

Required Attributes

None.

Optional Attributes

None.

Content

A sequence constructor.

Appears in

The `<xsl:analyze-string>` element.

Defined in

XSLT section 15, "Regular Expressions."

Example

Here is an example of <xsl:analyze-string> that uses a regular expression to convert U.S. and Canadian telephone numbers of the form 999-999-9999 to the form +1 (999) 999-9999:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- analyze-string1.xsl -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:for-each select="phonelist/phonenumber">
      <xsl:analyze-string select="."
        regex="([0-9]{{3}})-([0-9]{{3}})-([0-9]{{4}})">
        <xsl:matching-substring>
          <xsl:text>&#xA;+1 (</xsl:text>
          <xsl:value-of select="regex-group(1)"/>
          <xsl:text>) </xsl:text>
          <xsl:value-of select="regex-group(2)"/>
          <xsl:text>-</xsl:text>
          <xsl:value-of select="regex-group(3)"/>
        </xsl:matching-substring>
      </xsl:analyze-string>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Our stylesheet uses the regular expression ([0-9]{{3}})-([0-9]{{3}})-([0-9]{{4}}), which creates three regex-groups. (Each set of parentheses represents a regex-group.) When we use this stylesheet with this list of phone numbers:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- phonelist.xml -->
<phonelist>
  <phonenumber>919-555-1212</phonenumber>
  <phonenumber>(919) 555-1212</phonenumber>
  <phonenumber>212.555.1212</phonenumber>
  <phonenumber>617-555-1212</phonenumber>
  <phonenumber>+86 555-1212</phonenumber>
</phonelist>
```

we get these results:

```
+1 (919) 555-1212
+1 (617) 555-1212
```

Of the five phone numbers in our input document, only two matched the regular expression. This example of <xsl:analyze-string> contains only an <xsl:matching-substring> element; <xsl:analyze-string> can also contain an <xsl:non-matching-substring> element.

See Also

The [2.0] <xsl:analyze-string> and [2.0] <xsl:analyze-string> elements.

<xsl:message>

Sends a message. How the message is sent can vary from one XSLT processor to the next, but it's typically written to the standard output device. This element is useful for debugging stylesheets.

Category

Instruction.

Required Attributes

None.

Optional Attributes

`terminate`
: If this attribute has the value `yes`, the XSLT processor stops execution after issuing this message. The default value for this attribute is `no`; if the `<xsl:message>` doesn't terminate the processor, the message is sent and processing continues.

  [2.0] In XSLT 2.0, the value of the `terminate` attribute can be an attribute value template, allowing its value to be calculated at runtime.

[2.0] `select`
: An XPath expression that defines the content of this message.

Content

An XSLT template.

Appears in

`<xsl:message>` appears inside a template.

Defined in

[1.0] XSLT section 13, "Messages."

[2.0] XSLT section 17, "Messages."

Example

Here's a stylesheet that uses the `<xsl:message>` element to trace the transformation of an XML document. We'll use our list of recently purchased albums again:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
```

```
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

We'll list all of the purchased albums in an HTML table, with the background color of each row cycling through various colors. Our stylesheet uses `<xsl:message>` elements to indicate the background color of each row in the table:

```
<?xml version="1.0"?>
<!-- message.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>
          <xsl:value-of select="list/title"/>
        </title>
      </head>
      <body style="font-family: sans-serif; color: white;">
        <h1 style="color: black;">
          <xsl:value-of select="list/title"/>
        </h1>
        <table border="1" cellpadding="5"
          style="font-weight: bold;">
          <xsl:for-each select="list/listitem">
            <tr>
              <td>
                <xsl:attribute name="style">
                  <xsl:choose>
                    <xsl:when test="position() mod 4 = 0">
                      <xsl:text>background: yellow; color: black;</xsl:text>
                      <xsl:message terminate="no">
                        <xsl:text>Background color is yellow</xsl:text>
                      </xsl:message>
                    </xsl:when>
                    <xsl:when test="position() mod 4 = 1">
                      <xsl:text>background: blue;</xsl:text>
                      <xsl:message terminate="no">
                        <xsl:text>Background color is blue</xsl:text>
                      </xsl:message>
                    </xsl:when>
                    <xsl:when test="position() mod 4 = 2">
                      <xsl:text>background: white; color: black;</xsl:text>
                      <xsl:message terminate="no">
                        <xsl:text>Background color is white</xsl:text>
                      </xsl:message>
                    </xsl:when>
                    <xsl:otherwise>
                      <xsl:text>background: black;</xsl:text>
                      <xsl:message terminate="no">
                        <xsl:text>Background color is black</xsl:text>
                      </xsl:message>
```

```
                    </xsl:otherwise>
                  </xsl:choose>
                </xsl:attribute>
                <xsl:value-of select="."/>
              </td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

Note that the XSLT specification doesn't define how the message is issued. When we use this stylesheet with Saxon 8.8J, we get these results:

```
Background color is blue
Background color is white
Background color is black
Background color is yellow
Background color is blue
Background color is white
Background color is black
```

Running this stylesheet with Xalan gives us the same messages, along with information about which line in the stylesheet generated those messages:

```
file:///c:/message.xsl; Line #34; Column #51; Background color is blue
file:///c:/message.xsl; Line #40; Column #51; Background color is white
file:///c:/message.xsl; Line #46; Column #51; Background color is black
file:///c:/message.xsl; Line #28; Column #51; Background color is yellow
file:///c:/message.xsl; Line #34; Column #51; Background color is blue
file:///c:/message.xsl; Line #40; Column #51; Background color is white
file:///c:/message.xsl; Line #46; Column #51; Background color is black
```

[2.0] <xsl:namespace>

Allows you to create a namespace node in the result tree.

Category

Instruction.

Required Attribute

`name`
: The namespace prefix.

Optional Attribute

`select`
: An XPath expression that defines the value of the namespace itself. If you don't use the `select` attribute, the `<xsl:namespace>` element must contain content.

Content

The <xsl:namespace> element must have content or a select attribute; it is an error if it has both or neither. It is also an error if the <xsl:namespace> element evaluates to a zero-length string.

Appears in

<xsl:namespace> appears inside a template.

Defined in

[2.0] XSLT section 11.7, "Creating Namespace Nodes."

Example

Courtesy of the XSLT 2.0 spec, here is a stylesheet with a literal result element that contains an <xsl:namespace> element:

```
<?xml version="1.0"?>
<!-- namespace.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/">
    <data xsi:type="xs:integer"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <xsl:namespace name="xs"
        select="'http://www.w3.org/2001/XMLSchema'"/>
      <xsl:text>42</xsl:text>
    </data>
  </xsl:template>

</xsl:stylesheet>
```

Notice that the value of the select attribute is in single quotes inside double quotes; without the single quotes, this would select any <http://www.w3.org/2001/XMLSchema> elements, causing an error.

The results of this stylesheet look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<data xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xsi:type="xs:integer">42</data>
```

We could have written the stylesheet without the select attribute, putting the value of the namespace inside the <xsl:namespace> element:

```
<?xml version="1.0"?>
<!-- namespace2.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/">
    <data xsi:type="xs:integer"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <xsl:namespace name="xs">
        <xsl:text>http://www.w3.org/2001/XMLSchema</xsl:text>
      </xsl:namespace>
      <xsl:text>42</xsl:text>
    </data>
  </xsl:template>

</xsl:stylesheet>
```

Finally, we could generate the same results with a stylesheet that doesn't use `<xsl:namespace>` at all:

```
<?xml version="1.0"?>
<!-- namespace3.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/">
    <data xsi:type="xs:integer"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xsl:text>42</xsl:text>
    </data>
  </xsl:template>

</xsl:stylesheet>
```

The only time you're likely to use `<xsl:namespace>` is when you're creating a new element with `<xsl:element>` and you want to define a namespace on the new element, particularly if that namespace is used in content. (If that namespace is used in the names of elements or attributes, it would be copied to the result element automatically.)

<xsl:namespace-alias>

Allows you to define an alias for a namespace when using the namespace directly would complicate processing. This seldom-used element is the simplest way to write a stylesheet that generates another stylesheet.

Category

Top-level element.

Required Attributes

`result-prefix`
: Defines the prefix for the namespace referred to by the alias. This prefix must be declared in the stylesheet, regardless of whether any elements in the stylesheet use it.

`stylesheet-prefix`
: Defines the prefix used in the stylesheet to refer to the namespace.

[2.0] In XSLT 2.0, you can use the value `#default` for either the `result-prefix` or `stylesheet-prefix` attributes. As you would expect, it is an error if there is no default namespace. (A default namespace is defined with `xmlns=`.)

Optional Attributes

None.

Content

None. `<xsl:namespace-alias>` is an empty element.

Appears in

`<xsl:namespace-alias>` is a top-level element and can appear only as a child of `<xsl:stylesheet>`.

Defined in

[1.0] XSLT section 7.1.1, "Literal Result Elements."

[2.0] XSLT section 11.1.4, "Namespace Aliasing."

Example

As we mentioned before, this element is normally used to create an XSLT stylesheet that generates another stylesheet. Use `<xsl:namespace-alias>` when you want the output to contain an element or attribute that would normally be handled by the XSLT processor.

Our sample here creates a stylesheet that generates another stylesheet that copies any input document to the output. Here's our original stylesheet that uses `<xsl:namespace-alias>`:

```
<?xml version="1.0"?>
<!-- namespace-alias.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xslout="[anything but the XSL namespace]">

  <xsl:output method="xml" indent="yes"/>

  <xsl:namespace-alias stylesheet-prefix="xslout"
    result-prefix="xsl"/>

  <xsl:template match="/">
    <xslout:stylesheet version="1.0">
```

```
      <xslout:output method="xml"/>
      <xslout:template match="/">
        <xslout:copy-of select="."/>
      </xslout:template>
    </xslout:stylesheet>
  </xsl:template>

</xsl:stylesheet>
```

When we run this stylesheet with any XML document at all, we get a new stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <xsl:output method="xml"/>
   <xsl:template match="/">
      <xsl:copy-of select="."/>
   </xsl:template>
</xsl:stylesheet>
```

In our original stylesheet, we use `<xsl:namespace-alias>` to make sure the XSLT processor doesn't treat the XSLT elements we want to copy to the output stylesheet as instructions. Notice that when we define the namespace we'll be using as the alias, that namespace *cannot* be the same as the XSLT namespace. If it is, the XSLT processor will process our `<xslout:` elements as XSLT elements, defeating the purpose of using `<xsl:namespace-alias>`.

In the generated stylesheet, notice that the namespace value is the namespace associated with the `result-prefix` defined in the `<xsl:namespace-alias>` element. Be aware that the actual prefix in the generated stylesheet might not be `xsl` as it is here. Saxon generates a prefix of `xsl`, while Xalan uses the prefix `xslout` that we used in our original stylesheet. Regardless of the prefix, the generated stylesheet works the same.

[2.0] To illustrate using the default namespace with `<xsl:namespace-alias>`, we'll look at two stylesheets, one of which uses `result-prefix="#default"` and one of which uses `stylesheet-prefix="#default"`. Here is the first stylesheet:

```
<?xml version="1.0"?>
<!-- namespace-alias-2.xsl -->
<stylesheet version="1.0"
  xmlns="http://www.w3.org/1999/XSL/Transform"
  xmlns:xslout="[anything but the XSL namespace]">

  <output method="xml" indent="yes"/>

  <namespace-alias stylesheet-prefix="xslout"
    result-prefix="#default"/>

  <template match="/">
    <xslout:stylesheet version="1.0">
      <xslout:output method="xml"/>
      <xslout:template match="/">
        <xslout:copy-of select="."/>
      </xslout:template>
    </xslout:stylesheet>
  </template>
```

```
</stylesheet>
```

The output from this stylesheet looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<stylesheet xmlns="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <output method="xml"/>
   <template match="/">
      <copy-of select="."/>
   </template>
</stylesheet>
```

The original stylesheet and the generated stylesheet work because all of the XSLT elements are in the default namespace, `http://www.w3.org/1999/XSL/Transform`.

Here is the second stylesheet:

```
<?xml version="1.0"?>
<!-- namespace-alias-3.xsl -->
<xsl:stylesheet version="1.0"
  xmlns="[anything but the XSL namespace]"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes"/>

  <xsl:namespace-alias stylesheet-prefix="#default"
    result-prefix="xsl"/>

  <xsl:template match="/">
    <stylesheet version="1.0">
      <output method="xml"/>
      <template match="/">
        <copy-of select="."/>
      </template>
    </stylesheet>
  </xsl:template>

</xsl:stylesheet>
```

The output from this stylesheet looks more typical:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <xsl:output method="xml"/>
   <xsl:template match="/">
      <xsl:copy-of select="."/>
   </xsl:template>
</xsl:stylesheet>
```

[2.0] <xsl:next-match>

Tells the XSLT processor to select a template that is the next lowest in priority than the current template. This works similarly to `<xsl:apply-imports>`, although `<xsl:next-match>` works with all templates, including those in the main stylesheet and in any stylesheets included or

imported. Using <xsl:next-match> allows you to set up templates that work like overridden methods in object-oriented languages; using <xsl:next-match> is conceptually the same as calling super( ) in a Java program.

Category

Instruction.

Required Attributes

None.

Optional Attributes

None.

Content

Any number of <xsl:with-param> and <xsl:fallback> elements.

Appears in

<xsl:next-match> appears inside a template.

Defined in

[2.0] XSLT section 6.7, "Overriding Template Rules."

Example

For our example, we'll create some templates to process this HTML document:

```
<!-- element-discussion.html -->
<html>
  <head>
    <title>Interesting new XSLT elements</title>
  </head>
  <body>
    <h1>Interesting new XSLT elements</h1>
    <p>XSLT 2.0 has lots of interesting new elements.
    We'll mention a couple of them here. </p>
    <h1>The <code>&lt;xsl:next-match&gt;</code> element</h1>
    <p>One of the most interesting new elements in XSLT 2.0
    is <code>&lt;xsl:next-match&gt;</code>. </p>
    <h1>The <code>&lt;xsl:perform-sort&gt;</code> element</h1>
    <p>Don't forget about <code>&lt;xsl:perform-sort&gt;</code>,
    though.  It's very interesting as well.</p>
  </body>
</html>
```

Our example document has HTML <code> elements. Some of them are inside <h1> elements, others are inside <p> elements. To illustrate how <xsl:next-match> works, we'll create two different templates to process the <code> elements. The basic template puts the text of the element into a monospaced font; the overriding template changes the color of the <code> element if it occurs inside an <h1> element. Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- next-match.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="html">
    <html>
      <head>
        <title>
          <xsl:value-of select="head/title"/>
        </title>
      </head>
      <body style="font-family: sans-serif;">
        <xsl:apply-templates select="body/*"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="h1/code">
    <span style="color: red;">
      <xsl:next-match/>
    </span>
  </xsl:template>

  <xsl:template match="code">
    <span style="font-family: monospace;">
      <xsl:apply-templates select="*|text()"/>
    </span>
  </xsl:template>

  <xsl:template match="h1">
    <h1>
      <xsl:apply-templates select="*|text()"/>
    </h1>
  </xsl:template>

  <xsl:template match="p">
    <p>
      <xsl:apply-templates select="*|text()"/>
    </p>
  </xsl:template>

</xsl:stylesheet>
```

The key to using `<xsl:next-match>` is that the templates involved have different priorities. In our example here, the different priorities are set by the default XSLT precedence rules. A more specific rule always has precedence over a less specific one, so the highest priority rule for a `<code>` element inside an `<h1>` element is the `match="h1/code"` template. Using `<xsl:next-match>` inside that template invokes the next template in priority order—the template with `match="code"`.

The way our stylesheet is written, the default behavior of processing a `<code>` element is to create a `<span>` element that uses a monospaced font. The more specific (and therefore higher

priority) template creates a element that sets the font color to red. If we want to use a different font for all <code> elements, we would only have to change the match="code" template that sets the font-family property. Here are the results of our stylesheet:

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Interesting new XSLT elements</title>
   </head>
   <body style="font-family: sans-serif;">
      <h1>Interesting new XSLT elements</h1>
      <p>XSLT 2.0 has lots of interesting new elements.
         We'll mention a couple of them here.
      </p>
      <h1>The <span style="color: red;">
        <span style="font-family: monospace;">
        &lt;xsl:next-match&gt;</span>
        </span> element
      </h1>
      <p>One of the most interesting new elements in XSLT 2.0
         is <span style="font-family: monospace;">
        &lt;xsl:next-match&gt;</span>.
      </p>
      <h1>The <span style="color: red;">
        <span style="font-family: monospace;">
        &lt;xsl:perform-sort&gt;</span>
        </span> element
      </h1>
      <p>Don't forget about <span style="font-family: monospace;">
        &lt;xsl:perform-sort&gt;</span>,
        though.  It's very interesting as well.
      </p>
   </body>
</html>
```

The output looks like Figure A-11.

[2.0] <xsl:non-matching-substring>

Defines what to do when a string doesn't match a regular expression. The regular expression is defined on the regex attribute of the <xsl:analyze-string> element that contains the <xsl:non-matching-substring> element.

Category

Instruction (this is effectively part of the <xsl:analyze-string> element).

Required Attributes

None.

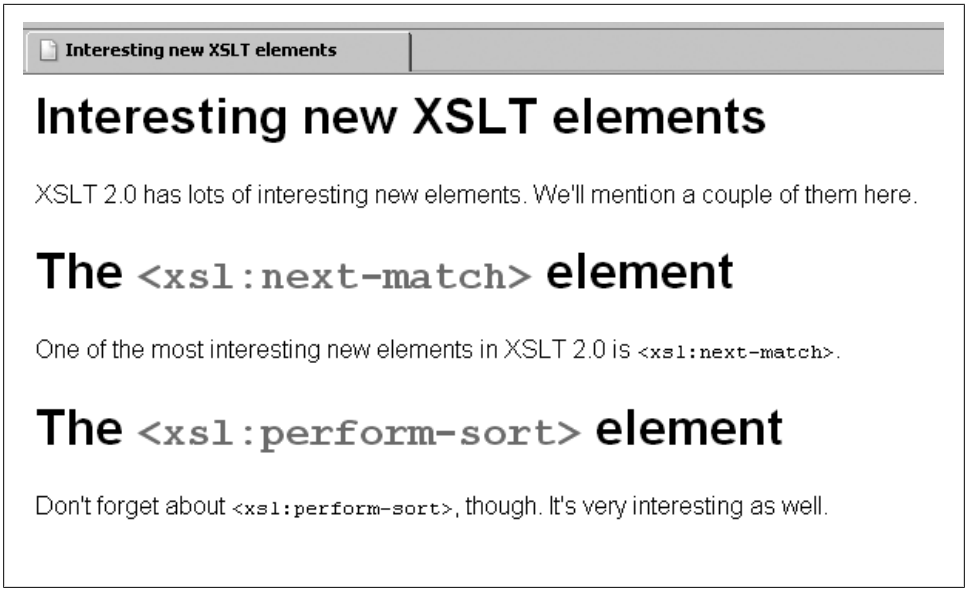


Figure A-11. HTML generated with <xsl:next-match>

Optional Attributes

None.

Content

A sequence constructor.

Appears in

The `<xsl:analyze-string>` element.

Defined in

XSLT section 15, "Regular Expressions."

Example

Here is an example of `<xsl:analyze-string>` that uses a regular expression to convert U.S. and Canadian telephone numbers of the form `999-999-9999` to the form `+1 (999) 999-9999`:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- non-matching-substring.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:for-each select="phonelist/phonenumber">
      <xsl:analyze-string select="."
```

```
        regex="([0-9]{{3}})-([0-9]{{3}})-([0-9]{{4}})">
        <xsl:non-matching-substring>
          <xsl:text>&#xA;   Unrecognized phone number: </xsl:text>
          <xsl:value-of select="."/>
        </xsl:non-matching-substring>
      </xsl:analyze-string>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Our stylesheet uses the regular expression `([0-9]{{3}})-([0-9]{{3}})-([0-9]{{4}})`. When we use this stylesheet with this list of phone numbers:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- phonelist.xml -->
<phonelist>
  <phonenumber>919-555-1212</phonenumber>
  <phonenumber>(919) 555-1212</phonenumber>
  <phonenumber>212.555.1212</phonenumber>
  <phonenumber>617-555-1212</phonenumber>
  <phonenumber>+86 555-1212</phonenumber>
</phonelist>
```

we get these results:

```
Unrecognized phone number: (919) 555-1212
Unrecognized phone number: 212.555.1212
Unrecognized phone number: +86 555-1212
```

This example of `<xsl:analyze-string>` contains only an `<xsl:non-matching-substring>` element; `<xsl:analyze-string>` can also contain an `<xsl:matching-substring>` element.

See Also

The [2.0] `<xsl:analyze-string>` element and the [2.0] `<xsl:matching-substring>` element.

<xsl:number>

Displays a number. It is most often used to number parts of a document, although it can also be used to format a numeric value.

Category

Instruction.

Required Attributes

None.

Optional Attributes

`count`
: The `count` attribute is an XPath pattern that defines what should be counted. If the `count` attribute isn't specified, it counts nodes with the same name as the current node.

`level`
: This attribute defines what levels of the source tree should be considered when numbering elements. The three valid values for this attribute are `single`, `multiple`, and `any`:

  `single`
  : Counts items at one level only. The XSLT processor goes to the first node in the `ancestor-or-self` axis that matches the `count` attribute, and then counts that node plus all its preceding siblings that also match the `count` attribute. This is the default value.

  `multiple`
  : Counts items at multiple levels. The XSLT processor looks at all the ancestors of the current node and at the current node itself, and then it selects all nodes that match the `count` attribute.

  `any`
  : Includes all of the current node's ancestors (as `level="multiple"` does) as well as all elements in the `preceding` axis.

  In all of these cases, if the `from` attribute is used, the only ancestors that are examined are descendants of the nearest ancestor that matches the `from` attribute. In other words, with `from="h1"`, the only nodes considered for counting are those that appear under the nearest `<h1>` attribute.

`from`
: The `from` attribute is an XPath pattern that defines where counting starts. For example, `from="h1"` means that counting should begin at the previous `<h1>` element.

`value`
: An expression that is converted to a number. Using this attribute is a quick way to format a number; the element `<xsl:number value="7" format="i:"/>` returns the string `"vii:"`.

  If the `value` attribute is specified, none of the `count`, `level`, `from`, or `select` attributes may be used.

`format`
: The `format` attribute defines the format of the generated number:

  `format="1"`
  : Formats a sequence of numbers as `1 2 3 4 5 6 7 8 9 10 11 ...`.

`format="01"`
: Formats a sequence of numbers as `01 02 03 04 ... 09 10 11 ... 99 100 101 ...`.

`format="a"`
: Formats a sequence of numbers as `a b c d e f ... x y z aa ab ac ...`.

`format="A"`
: Formats a sequence of numbers as `A B C D E F ... X Y Z AA AB AC ...`.

`format="i"`
: Formats a sequence of numbers as `i ii iii iv v vi vii viii ix x ...`.

`format="I"`
: Formats a sequence of numbers as `I II III IV V VI VII VIII IX X ...`.

[2.0] `format="w"`
: Formats a sequence of numbers as words. When used with `ordinal="yes"` and `lang="en"`, it formats a sequence of numbers as `first second third fourth ...`. If `ordinal="yes"` is not specified, it formats a sequence of numbers as `one two three four ...`. If a given `format`, `ordinal`, and `lang` combination is not supported, the XSLT processor is required to format the numbers as `1 2 3 4 5 ...` as a last resort.

[2.0] `format="Ww"`
: Formats a sequence of numbers as words. When used with `ordinal="yes"` and `lang="en"`, it formats a sequence of numbers as `First Second Third Fourth ...`. If `ordinal="yes"` is not specified, it formats a sequence of numbers as `One Two Three Four ...`. If a given `format`, `ordinal`, and `lang` combination is not supported, the XSLT processor is required to format the numbers as `1 2 3 4 5 ...` as a last resort.

`format="anything else"`
: How this works depends on the XSLT processor you're using. The XSLT specification lists several other numbering schemes (`format="&#0E51;"` for Thai numbering, for example); check your XSLT processor's documentation to see which formats it supports. If the XSLT processor doesn't support the numbering scheme you requested, the XSLT spec requires that it use `format="1"` as the default.

[2.0] `lang`
: The `lang` attribute defines the language whose alphabet should be used. Different XSLT processors support different language values, so check the documentation of your favorite XSLT processor for more information.

`letter-value`
: This attribute has the value `alphabetic` or `traditional`. There are a number of languages in which two letter-based numbering schemes are used; one assigns numeric values in alphabetic sequence, while the other uses a tradition native to that

language. (Roman numerals—a letter-based numbering scheme that doesn't use an alphabetic order—are one example.) The default for this attribute is `alphabetic`.

`grouping-separator`
: This attribute is the character that should be used between groups of digits in a generated number. The default is the comma (`,`).

`grouping-size`
: This attribute defines the number of digits that appear in each group; the default is `3`.

[2.0] `ordinal`
: Defines that ordinal numbers should be used; valid values are `yes` and `no`. The `ordinal`, `format`, and `lang` attributes work together. For example, `ordinal="yes"`, `format="1"`, and `lang="en"` produces the sequence `1st 2nd 3rd 4th`. With `format="w"` the sequence is `first second third fourth`. `format="Ww"` produces `First Second Third Fourth`. If the combination of `ordinal`, `format`, and `lang` is not supported, the XSLT processor is required to generate cardinal numbers (`1 2 3 4`) instead. See the documentation for your XSLT processor to find out which language and format combinations it supports for the `ordinal` attribute.

[2.0] `select`
: Selects a node to be numbered. This allows you to select a node other than the context node.

Content

None. `<xsl:number>` is an empty element.

Appears in

`<xsl:number>` appears inside a template.

Defined in

[1.0] XSLT section 7.7, "Numbering."

[2.0] XSLT section 12, "Numbering."

Example

To fully illustrate how `<xsl:number>` works, we'll need an XML document with many things to count. Here's the document we'll use:

```
<?xml version="1.0"?>
<!-- items-to-number.xml -->
<book>
  <chapter><title>Alfa Romeo</title>
    <sect1><title>Bentley</title></sect1>
    <sect1><title>Chevrolet</title>
      <sect2><title>Dodge</title>
        <sect3><title>Eagle</title></sect3>
      </sect2>
    </sect1>
```

```
  </chapter>
  <chapter><title>Ford</title>
    <sect1><title>GMC</title>
      <sect2><title>Honda</title>
        <sect3><title>Isuzu</title></sect3>
        <sect3><title>Javelin</title></sect3>
        <sect3><title>K-Car</title></sect3>
        <sect3><title>Lincoln</title></sect3>
      </sect2>
      <sect2><title>Mercedes</title></sect2>
      <sect2><title>Nash</title>
        <sect3><title>Opel</title></sect3>
        <sect3><title>Pontiac</title></sect3>
      </sect2>
      <sect2><title>Quantum</title>
        <sect3><title>Rambler</title></sect3>
        <sect3><title>Studebaker</title></sect3>
      </sect2>
    </sect1>
    <sect1><title>Toyota</title></sect1>
  </chapter>
</book>
```

We'll use `<xsl:number>` in several different ways to illustrate the various options we have in numbering things. We'll look at several short stylesheets and their results. Here's the first one:

```
<?xml version="1.0"?>
<!-- number1.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="book">
    <xsl:for-each select="chapter|.//sect1|.//sect2|.//sect3">
      <xsl:number level="multiple" count="chapter|sect1|sect2|sect3"
          format="1.1.1.1. "/>
        <xsl:value-of select="title"/>
        <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Here are our results:

```
1. Alfa Romeo
1.1. Bentley
1.2. Chevrolet
1.2.1. Dodge
1.2.1.1. Eagle
2. Ford
2.1. GMC
2.1.1. Honda
2.1.1.1. Isuzu
2.1.1.2. Javelin
2.1.1.3. K-Car
```

```
2.1.1.4. Lincoln
2.1.2. Mercedes
2.1.3. Nash
2.1.3.1. Opel
2.1.3.2. Pontiac
2.1.4. Quantum
2.1.4.1. Rambler
2.1.4.2. Studebaker
2.2. Toyota
```

Here we use `level="multiple"` to count the `<chapter>`, `<sect1>`, `<sect2>`, and `<sect3>` elements. Numbering these at multiple levels gives us a dotted decimal number for each element. We can look at the number next to `Studebaker` and know that it is the second `<sect3>` element inside the fourth `<sect2>` element inside the first `<sect1>` element inside the second `<chapter>` element.

In the stylesheet, we use `<xsl:for-each>` to select the items we want to list, and we use `<xsl:number>` to number them. As we'll see in a minute, we don't have to select all the items in the `<xsl:for-each>` for them to be numbered correctly. If we wrote an XPath expression that selected just the `<sect3>` element with `<title>Studebaker</title>`, the output would be `2.1.4.2 Studebaker`, just as it is earlier.

Our next stylesheet uses `level="any"` to count all of the `<chapter>`, `<sect1>`, `<sect2>`, and `<sect3>` elements in order. The stylesheet looks like this:

```
<?xml version="1.0"?>
<!-- number2.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="book">
    <xsl:for-each select="chapter|.//sect1|.//sect2|.//sect3">
      <xsl:number level="any" count="chapter|sect1|sect2|sect3"
        format="1. "/>
        <xsl:value-of select="title"/>
        <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

The results are straightforward:

```
1. Alfa Romeo
2. Bentley
3. Chevrolet
4. Dodge
5. Eagle
6. Ford
7. GMC
8. Honda
9. Isuzu
10. Javelin
11. K-Car
```

```
12. Lincoln
13. Mercedes
14. Nash
15. Opel
16. Pontiac
17. Quantum
18. Rambler
19. Studebaker
20. Toyota
```

Stylesheet *number3.xsl* uses `level="single"` to count the elements at each level. This means that the fourth `<sect3>` element inside a given `<sect2>` element will be numbered with a `4` (or `iv` or `D` or whatever the appropriate value would be). Notice that the number used for each element is the same as the last number beside each element in Test 1.

```
<?xml version="1.0"?>
<!-- number3.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="book">
    <xsl:for-each select="chapter|.//sect1|.//sect2|.//sect3">
      <xsl:number level="single" count="chapter|sect1|sect2|sect3"
        format="1.1.1.1. "/>
        <xsl:value-of select="title"/>
        <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

These results are less useful, but here they are:

```
1. Alfa Romeo
1. Bentley
2. Chevrolet
1. Dodge
1. Eagle
2. Ford
1. GMC
1. Honda
1. Isuzu
2. Javelin
3. K-Car
4. Lincoln
2. Mercedes
3. Nash
1. Opel
2. Pontiac
4. Quantum
1. Rambler
2. Studebaker
2. Toyota
```

Stylesheet *number4.xsl* does a couple of things differently: first, it uses the uppercase-alpha and lowercase-roman numbering styles. Second, it counts elements at multiple levels (for the `<chapter>`, `<sect1>`, and `<sect2>` elements), but we process only the `<sect2>` elements. Even though we output only the title text for the `<sect2>` elements, we can still generate the appropriate multilevel numbers:

```
<?xml version="1.0"?>
<!-- number4.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="book">
    <xsl:for-each select=".//sect2">
      <xsl:number level="multiple" count="chapter|sect1|sect2"
        format="I-A-i: "/>
        <xsl:value-of select="title"/>
        <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Because we're numbering only the `<sect2>` elements, our output is much shorter:

```
I-B-i: Dodge
II-A-i: Honda
II-A-ii: Mercedes
II-A-iii: Nash
II-A-iv: Quantum
```

As we mentioned before, the elements we selected with the `<xsl:for-each>` element are numbered correctly, even though we don't reference the elements around them.

number5.xsl generates numbers similarly to *number4.xsl*, except that it uses the `from` attribute. We generate numbers for `<sect3>` elements in four stages. First, we count the `<chapter>` ancestors, starting at the first `<book>` ancestor; then we count the `<sect1>` ancestors, starting at the first `<chapter>` ancestor, etc.:

```
<?xml version="1.0"?>
<!-- number5.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="book">
    <xsl:for-each select=".//sect3">
      <xsl:number level="any" from="book" count="chapter" format="1."/>
      <xsl:number level="any" from="chapter" count="sect1" format="1."/>
      <xsl:number level="any" from="sect1" count="sect2" format="1."/>
      <xsl:number level="any" from="sect2" count="sect3" format="1. "/>
      <xsl:value-of select="title"/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
```

```
  </xsl:template>

</xsl:stylesheet>
```

Our results are similar to those for *number4.xsl*:

```
1.2.1.1. Eagle
2.1.1.1. Isuzu
2.1.1.2. Javelin
2.1.1.3. K-Car
2.1.1.4. Lincoln
2.1.3.1. Opel
2.1.3.2. Pontiac
2.1.4.1. Rambler
2.1.4.2. Studebaker
```

For stylesheet *number6.xsl* we start counting at 1000. We use the `grouping-separator` attribute here as well:

```
<?xml version="1.0"?>
<!-- number6.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="book">
    <xsl:for-each select="chapter|.//sect1|.//sect2|.//sect3">
      <xsl:variable name="value1">
        <xsl:number level="any" count="chapter|sect1|sect2|sect3"/>
      </xsl:variable>
      <xsl:number value="$value1 + 999"
        grouping-separator="," grouping-size="3"/>
      <xsl:text>. </xsl:text>
      <xsl:value-of select="title"/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

To start counting at 1000, we use a variable to store the number generated by `<xsl:number>`. Next we use `<xsl:number>` again to output that value plus 999. The output looks like this:

```
1,000. Alfa Romeo
1,001. Bentley
1,002. Chevrolet
1,003. Dodge
1,004. Eagle
1,005. Ford
1,006. GMC
1,007. Honda
1,008. Isuzu
1,009. Javelin
1,010. K-Car
1,011. Lincoln
1,012. Mercedes
```

```
1,013. Nash
1,014. Opel
1,015. Pontiac
1,016. Quantum
1,017. Rambler
1,018. Studebaker
1,019. Toyota
```

In stylesheet *number7.xsl*, we number items from the first and second `<sect1>` elements (`<sect1>` elements whose `position( )` is less than 3) in the second `<chapter>` element:

```
<?xml version="1.0"?>
<!-- number7.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="book">
    <xsl:for-each select="chapter[2]/sect1[position() < 3]">
      <xsl:for-each select="chapter|.//sect1|.//sect2|.//sect3">
        <xsl:number level="multiple" count="chapter|sect1|sect2|sect3"
          format="1.1.1.1. "/>
        <xsl:value-of select="title"/>
        <xsl:text>&#xA;</xsl:text>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

The results look like this:

```
2.1.1. Honda
2.1.1.1. Isuzu
2.1.1.2. Javelin
2.1.1.3. K-Car
2.1.1.4. Lincoln
2.1.2. Mercedes
2.1.3. Nash
2.1.3.1. Opel
2.1.3.2. Pontiac
2.1.4. Quantum
2.1.4.1. Rambler
2.1.4.2. Studebaker
```

This style of numbering is useful for a partial table of contents.

[2.0] Now we'll look at some of the new features added in XSLT 2.0. To keep the output short, we'll only count the `<sect2>` elements as we did in stylesheet *number4.xsl*. First, we'll use the `ordinal` attribute:

```
<?xml version="1.0"?>
<!-- number8.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
  <xsl:output method="text"/>

  <xsl:template match="book">
    <xsl:for-each select=".//sect2">
      <xsl:number level="any" count="chapter|sect1|sect2|sect3"
        format="Ww - " ordinal="yes"/>
        <xsl:value-of select="title"/>
        <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

The wordy output looks like this:

```
Fourth - Dodge
Eighth - Honda
Thirteenth - Mercedes
Fourteenth - Nash
Seventeenth - Quantum
```

Now we'll use the combination of `format` and `lang`:

```
<?xml version="1.0"?>
<!-- number9.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="book">
    <xsl:for-each select=".//sect2">
      <xsl:number level="any" count="chapter|sect1|sect2|sect3"
        format="w - " lang="de"/>
        <xsl:value-of select="title"/>
        <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Using this stylesheet with Saxon, our cardinal numbers appear in German:

```
vier - Dodge
acht - Honda
dreizehn - Mercedes
vierzehn - Nash
siebzehn - Quantum
```

If we request a combination of `format`, `ordinal`, and `lang` that the XSLT processor doesn't support, the processor reverts to its default behavior. We'll use *number10.xsl* to see what Saxon does when we ask for cardinal numbers in Polish:

```
<?xml version="1.0"?>
<!-- number10.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
  <xsl:output method="text"/>

  <xsl:template match="book">
    <xsl:for-each select=".//sect2">
      <xsl:number level="any" count="chapter|sect1|sect2|sect3"
        format="w - " lang="pl"/>
        <xsl:value-of select="title"/>
        <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Saxon doesn't support Polish, so we get our numbers in English:

```
four - Dodge
eight - Honda
thirteen - Mercedes
fourteen - Nash
seventeen - Quantum
```

For our last example, we'll use `format="๑"` (Thai numbering) and `ordinal="yes"`:

```
<?xml version="1.0"?>
<!-- number11.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="book">
    <html>
      <head>
        <title>Thai numbering</title>
      </head>
      <body style="font-family: sans-serif;">
        <h1>Thai numbering</h1>
        <p style="font-size: 150%">
          <xsl:for-each select=".//sect2">
            <xsl:number level="any" count="sect2"
              format="&#x0E51; "/>
            <xsl:value-of select="title"/>
            <br/>
          </xsl:for-each>
        </p>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates an HTML document that counts all of the `<sect2>` elements and uses Thai numbering:

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

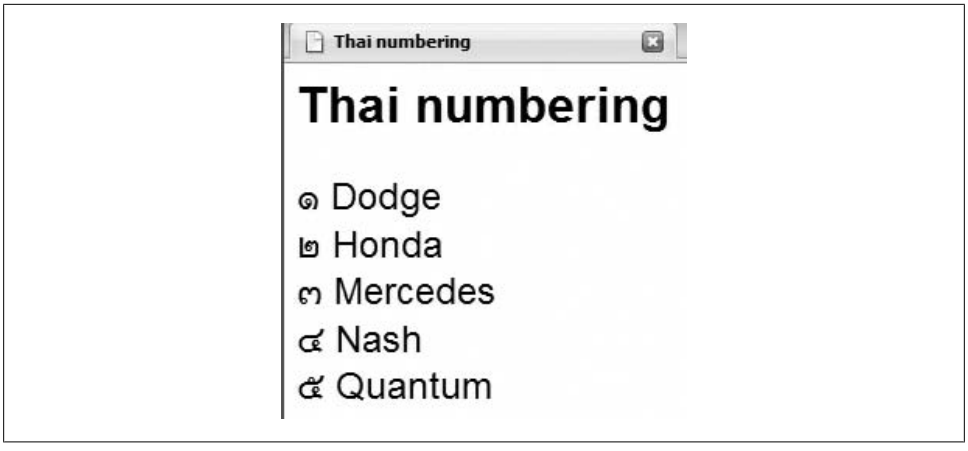


Figure A-12. A list with Thai numbering

```
        <title>Thai numbering</title>
      </head>
      <body style="font-family: sans-serif;">
        <h1>Thai numbering</h1>
        <p style="font-size: 150%">&#xOE51; Dodge<br>
        &#xOE52; Honda<br>&#xOE53; Mercedes<br>
        &#xOE54; Nash<br>&#xOE55; Quantum<br></p>
      </body>
    </html>
```

When formatted in a browser, the document appears as in Figure A-12.

<xsl:otherwise>

Defines the `else` or `default` case in an `<xsl:choose>` element. This element always appears inside an `<xsl:choose>` element, and it must always appear last.

Category

Subinstruction (`<xsl:otherwise>` always appears as part of an `<xsl:choose>` element).

Required Attributes

None.

Optional Attributes

None.

Content

A template.

Appears in

The <xsl:choose> element.

Defined in

[1.0] XSLT section 9.2, "Conditional Processing with xsl:choose."

[2.0] XSLT section 8.2, "Conditional Processing with xsl:choose."

Example

Here's an example that uses <xsl:choose> to select the background and foreground colors for the rows of an HTML table. We cycle among four different values, using <xsl:otherwise> to determine the default value of the style attribute in the generated HTML document. Here's the XML document we'll use:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en_GB">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

And here's our stylesheet:

```
<?xml version="1.0"?>
<!-- otherwise.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>
          <xsl:value-of select="list/title"/>
        </title>
      </head>
      <body style="font-family: sans-serif; color: white;">
        <h1 style="color: black;">
          <xsl:value-of select="list/title"/>
        </h1>
        <table border="1" cellpadding="5"
          style="font-weight: bold;">
          <xsl:for-each select="list/listitem">
            <tr>
              <td>
                <xsl:attribute name="style">
                  <xsl:choose>
```

```
                    <xsl:when test="position() mod 4 = 0">
                      <xsl:text>background: yellow; color: black;</xsl:text>
                    </xsl:when>
                    <xsl:when test="position() mod 4 = 1">
                      <xsl:text>background: blue;</xsl:text>
                    </xsl:when>
                    <xsl:when test="position() mod 4 = 2">
                      <xsl:text>background: white; color: black;</xsl:text>
                    </xsl:when>
                    <xsl:otherwise>
                      <xsl:text>background: black;</xsl:text>
                    </xsl:otherwise>
                  </xsl:choose>
                </xsl:attribute>
                <xsl:value-of select="."/>
              </td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

Here's the generated HTML document, which cycles through the various background and foreground colors:

```
<html>
  <head>
    <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Albums I've bought recently:</title>
  </head>
  <body style="font-family: sans-serif; color: white;">
    <h1 style="color: black;">Albums I've bought recently:</h1>
    <table border="1" cellpadding="5" style="font-weight: bold;">
      <tr>
        <td style="background: blue;">The Sacred Art of Dub</td>
      </tr>
      <tr>
        <td style="background: white; color: black;">Only the Poor Man Feel It</td>
      </tr>
      <tr>
        <td style="background: black;">Excitable Boy</td>
      </tr>
      <tr>
        <td style="background: yellow; color: black;">Aki Special</td>
      </tr>
      ...
    </table>
  </body>
</html>
```

When rendered, our HTML document looks like Figure A-13.

Figure A-13. Using <xsl:when> and <xsl:otherwise> to cycle among background colors

Notice that every fourth row has a style of `"background:black;"`. That value is generated by the `<xsl:otherwise>` element.

<xsl:output>

Defines the characteristics of an output document.

Category

Top-level element.

Required Attributes

None.

Optional Attributes

`method`

Typically has one of four values: `xml`, `html`, `xhtml`, or `text`. This value indicates the type of document that is generated. An XSLT processor can add other values to this list; how those values affect the generated document is determined by the XSLT processor. The default value is `xml` unless the root element of the output document is `<html>`.

[1.0] In XSLT 1.0, there are only three output methods defined in the standard: `xml`, `html`, and `text`. *[2.0]* In XSLT 2.0, `xhtml` has been added as an output method that must be supported by an XSLT processor. Individual processors are still free to add other values as well. To simplify our discussion here, we'll discuss `xhtml` as

one of the standard output types without constantly reminding you that it's not available in XSLT 1.0.

`version`
: Defines the version of the XML, HTML, or XHTML output document. This attribute is ignored with `method="text"`.

The `version` attribute of the `<xsl:output>` element works differently from the `version` attribute that is legal on every XSLT element in XSLT 2.0. See "[2.0] Attributes common to all XSLT elements" earlier in this appendix for more details.

`encoding`
: Defines the value of the `encoding` specified in the XML declaration in the output document. This attribute is ignored with `method="text"`.

`omit-xml-declaration`
: Defines whether the XML declaration is omitted in the output document. Allowable values are `yes` and `no`; the default is `no`. This attribute is used only with `method="xml"` and `method="xhtml"`.

`standalone`
: Defines the value of the `standalone` attribute of the XML declaration in the output document. Valid values are `yes` and `no`. This attribute is used only with `method="xml"` and `method="xhtml"`.

  [2.0] In XSLT 2.0, the value `omit` is also allowed. Coding `standalone="omit"` means the output document will not have a value for `standalone`.

`doctype-public`
: Defines the value of the `PUBLIC` attribute of the `DOCTYPE` declaration in the output document. This attribute defines the public identifier of the output document's DTD. This attribute is ignored with `method="text"`.

`doctype-system`
: Defines the value of the `SYSTEM` attribute of the `DOCTYPE` declaration in the output document. It defines the system identifier of the output document's DTD. This attribute is ignored with `method="text"`.

`cdata-section-elements`
: Lists the elements whose content should be written as `CDATA` sections in the output document. All restrictions and escaping conventions of `CDATA` sections are handled by the XSLT processor. If you need to list more than one element, separate the element names with one or more whitespace characters. This attribute is used only with `method="xml"` and `method="xhtml"`.

`indent`
: Specifies whether the tags in the output document should be indented. Allowable values are `yes` and `no`. This attribute is used only with `method="xml"`,

`method="html"`, or `method="xhtml"`, and the XSLT processor is not required to honor it. The default value is `yes` for `method="html"` and `method="xhtml"`, and `no` for `method="xml"`.

`media-type`
: Defines the MIME type of the output document. The default value is `text/xml` for `method="xml"`, `text/html` for `method="html"` and `method="xhtml"`, and `text/plain` for `method="text"`.

[2.0] `name`
: Names this output specification. The new [2.0] `<xsl:result-document>` element can use its `format` attribute to refer to this name. If your stylesheet uses `<xsl:result-document>` to create multiple output documents, you can define an output specification with the `name` attribute and use it wherever you need it.

[2.0] `byte-order-mark`
: Defines whether or not a byte-order mark is written to the output. Valid values are `yes` and `no`. The default is `yes` if the `encoding` is `UTF-16`, but the default for `UTF-8` varies from one XSLT processor to the next. The default is `no` for every other encoding.

[2.0] `escape-uri-attributes`
: Defines whether HTML and XHTML attributes with URI values should have special characters replaced with their hex equivalents. For example, if a URI contains spaces, the escaped value of the URI replaces each spaces with `%20`.

[2.0] `include-content-type`
: Defines whether the content type should be written to the output document. Valid values are `yes` and `no`; the default is `yes`. As an example, if `include-content-type` is used for `method="html"`, the element `<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">` is added to the `<head>` element of the HTML document.

[2.0] `normalization-form`
: Defines how Unicode characters should be normalized. Valid values are `NFC`, `NFD`, `NFKC`, `NFKD`, `fully-normalized`, and `none`. XSLT processors are free to support other values as well; see your processor's documentation to find out whether it supports any other normalization forms.

[2.0] `undeclare-prefixes`
: Defines whether the output document should include namespace undeclarations (an *undeclaration* associates a namespace prefix with an empty string, as in `xmlns:doug=""`). Valid values are `yes` and `no`, and this attribute has meaning only with `method="xml"` and when the `version` attribute is `1.1` or higher. See section 20 of the XSLT 2.0 specification for the complete details on this obscure attribute.

[2.0] `use-character-maps`
: Defines a space-separated list of named character maps (created with the [2.0] `<xsl:character-map>` that should be used when creating the output document).

Content

None. <xsl:output> is an empty element.

Appears in

<xsl:output> is a top-level element and can only appear as a child of <xsl:stylesheet>.

Defined in

[1.0] XSLT section 16, "Output."

[2.0] XSLT section 20, "Serialization."

Example

To illustrate the four output methods defined in the XSLT specification, we'll create four stylesheets, each of which uses one of the four methods. We'll use the following XML document in all four examples:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

We'll now look at our stylesheets and the results produced by each. First, let's look at the method="xml" stylesheet:

```
<?xml version="1.0"?>
<!-- output1.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes" encoding="ISO-8859-1"/>

  <xsl:template match="/">
    <catalog>
      <xsl:for-each select="/list/listitem">
        <album>
          <xsl:apply-templates/>
        </album>
      </xsl:for-each>
    </catalog>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates the following results:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<catalog>
   <album>The Sacred Art of Dub</album>
   <album>Only the Poor Man Feel It</album>
   <album>Excitable Boy</album>
   <album>Aki Special</album>
   <album>Combat Rock</album>
   <album>Talking Timbuktu</album>
   <album>The Birth of the Cool</album>
</catalog>
```

The output document has the `encoding` we specified in our stylesheet, and the output document has been indented. (Specifying `indent="no"` generates an XML document that's unreadable, although well-formed.)

Now let's look at the HTML version:

```
<?xml version="1.0"?>
<!-- output2.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" encoding="ISO-8859-1"/>

  <xsl:template match="/">
    <html>
      <head>
        <title><xsl:value-of select="/list/title"/></title>
      </head>
      <body>
        <h1><xsl:value-of select="/list/title"/></h1>
        <p>
          <xsl:for-each select="/list/listitem">
            <xsl:number format="1. "/>
            <xsl:value-of select="."/>
            <br/>
          </xsl:for-each>
        </p>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

Here is the HTML document generated by this stylesheet:

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
      <title>Albums I've bought recently:</title>
   </head>
   <body>
      <h1>Albums I've bought recently:</h1>
      <p>1. The Sacred Art of Dub<br>2. Only the Poor Man
Feel It<br>3. Excitable Boy<br>4. Aki Special<br>5.
Combat Rock<br>6. Talking Timbuktu<br>7. The Birth of the
```

```
Cool<br></p>
   </body>
</html>
```

(We added line breaks to make the listing legible.) Notice that the XSLT processor has automatically inserted a <META> element in the <head> of our HTML document. The <br> elements are old-fashioned <br> tags. Even though this style of XSLT output results in a document that is not well-formed XML (or XHTML), the document works with existing HTML browsers.

To generate valid XHTML, we'll use method="xhtml":

```
<?xml version="1.0"?>
<!-- output3.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output
    method="xhtml"
    encoding="ISO-8859-3"
    doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
    doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"/>

  <xsl:template match="/">
    <html>
      <head>
        <title><xsl:value-of select="/list/title"/></title>
      </head>
      <body>
        <h1><xsl:value-of select="/list/title"/></h1>
        <p>
          <xsl:for-each select="/list/listitem">
            <xsl:number format="1. "/>
            <xsl:value-of select="."/>
            <br/>
          </xsl:for-each>
        </p>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

Our results look like this:

```
<?xml version="1.0" encoding="ISO-8859-3"?>
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org
/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
   <head>
      <title>Albums I've bought recently:</title>
   </head>
   <body>
      <h1>Albums I've bought recently:</h1>
      <p>1. The Sacred Art of Dub
         <br></br>2. Only the Poor Man Feel It
```

```
        <br></br>3. Excitable Boy
        <br></br>4. Aki Special
        <br></br>5. Combat Rock
        <br></br>6. Talking Timbuktu
        <br></br>7. The Birth of the Cool
        <br></br>
      </p>
    </body>
</html>
```

(We added a line break in the `DOCTYPE` declaration; originally it appeared on a single long line.) Our `<br>` elements are well-formed XML now.

Our final stylesheet uses `method="text"`:

```
<?xml version="1.0"?>
<!-- output4.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"
    encoding="ISO-8859-3"
    indent="yes"
    omit-xml-declaration="no"
    standalone="yes"
    doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
    doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"/>

  <xsl:template match="/">
    <html>
      <head>
        <title><xsl:value-of select="/list/title"/></title>
      </head>
      <body>
        <h1><xsl:value-of select="/list/title"/></h1>
        <p>
          <xsl:for-each select="/list/listitem">
            <xsl:number format="1. "/>
            <xsl:value-of select="."/>
            <br/>
          </xsl:for-each>
        </p>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results, such as they are, from this stylesheet:

```
Albums I've bought recently:Albums I've bought recently:1. The Sacred
Art of Dub2. Only the Poor Man Feel It3. Excitable Boy4. Aki Special5.
Combat Rock6. Talking Timbuktu7. The Birth of the Cool
```

(As before, we inserted line breaks so the document would fit on the page.) These results are basically worthless. Why weren't our carefully coded HTML elements and all the information we put on the `<xsl:output>` element written to the text document? The reason is that the

method="text" outputs only text nodes to the result tree, and it ignores most of the attributes of <xsl:output>. Even though we requested that various HTML elements be generated along the way, they're ignored because we specified method="text".

[2.0] <xsl:output-character>

Defines a symbol and a string that should replace it. This is similar to an XML <!ENTITY> declaration.

Category

Declaration (this is effectively part of the <xsl:character-map> element).

Required Attributes

character
: The character to be replaced.

string
: The string that replaces the character.

Optional Attributes

None.

Content

None. <xsl:output-character> is an empty element.

Appears in

The <xsl:character-map> element.

Defined in

XSLT section 20, "Serialization."

Example

We'll repeat a small section of our example from the description of the <xsl:character-map> element. Here is a stylesheet that replaces two circled number characters (Unicode code points ૜ and ૝) with graphics:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- character-map2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" use-character-maps="circles"/>

  <xsl:character-map name="circles">
    <xsl:output-character character="&#x2780;"
      string="&lt;img src='images/circle1.gif'
```

```
      width='28' height='28'/&gt;"/>
    <xsl:output-character character="&#x2781;"
      string="&lt;img src='images/circle2.gif'
      width='28' height='28'/&gt;"/>
  </xsl:character-map>

  <xsl:template match="char-test">
    <html>
      <head>
        <title>A test of some special characters</title>
      </head>
      <body style="font-family: sans-serif;">
        <h1>A test of some special characters</h1>
        <xsl:apply-templates select="*"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="special-char">
    <p style="font-size: 200%;">
      <xsl:text>Here's a special character: </xsl:text>
      <xsl:value-of select="."/>
    </p>
  </xsl:template>

</xsl:stylesheet>
```

We'll transform this tabbed code listing:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- special-characters2.xml -->
<char-test>
  <special-char>&#x2780;</special-char>
  <special-char>&#x2781;</special-char>
</char-test>
```

Our result document has replaced the circled numbers with graphics:

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>A test of some special characters</title>
   </head>
   <body style="font-family: sans-serif;">
      <h1>A test of some special characters</h1>
      <p style="font-size: 200%;">Here's a special character:
      <img src='images/circle1.gif' width='28' height='28'/></p>
      <p style="font-size: 200%;">Here's a special character:
      <img src='images/circle2.gif'width='28' height='28'/></p>
   </body>
</html>
```

This is a useful way of replacing characters that aren't displayable on every browser. Our HTML document appears as in Figure A-14.

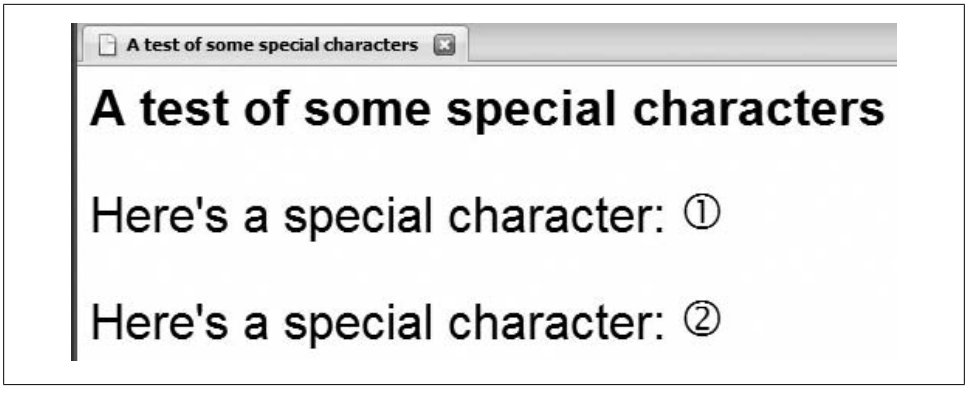

A test of some special characters
A test of some special characters
Here's a special character: ①
Here's a special character: ②

Figure A-14. HTML document generated with character maps

<xsl:param>

Defines the name and value of a parameter to be used in a stylesheet. This element can appear as a top-level element or inside the `<xsl:template>` element. If the `<xsl:param>` appears as a top-level element, it is a global parameter, visible to all areas of the stylesheet. The default value of the parameter can be defined in one of two ways: specified in the `select` attribute or defined inside the `<xsl:param>` element itself. *[2.0]* In XSLT 2.0, `<xsl:param>` can also appear inside the new `<xsl:function>` element.

Category

Instruction.

Required Attribute

`name`
: Defines the name of this parameter.

Optional Attributes

`select`
: Contains an XPath expression that defines the value of this parameter.

[2.0 – Schema] `as`
: Defines the datatype of this parameter. The datatype can be any of the built-in types or, if you have a schema-aware XSLT processor, a datatype defined in an XML Schema. For example, `as="xs:integer"` specifies a parameter that is a single integer, and `as="xs:string*"` specifies a parameter that is a sequence of zero or more strings. With a schema-aware XSLT processor, `as="schema-element(po:purchase-order)"` means that the parameter must be a valid `po:purchase-order`.

[2.0] `required`

Defines whether this parameter is required. Valid values are `yes` and `no`. If this is a parameter for an `<xsl:function>`, this attribute must not be defined. For stylesheet and template parameters, the default value is that the parameter is optional (`required="no"`).

It is a static error if an `<xsl:param>` element has a `required` attribute and a default value. Required parameters cannot have a `select` attribute, and they must be empty.

It is also a static error to use the `required` attribute on a parameter defined in an `<xsl:function>` element.

[2.0] `tunnel`

Defines whether this is a tunnel parameter. Valid values are `yes` and `no`, with `no` being the default. (For an in-depth discussion of tunnel parameters, see "Tunnel parameters" in Chapter 5.)

Content

If the `select` attribute is used, `<xsl:param>` must be empty. Otherwise, it contains an XSLT template.

[2.0] In XSLT 2.0, this rule is enforced much more strictly. It is a static error if a nonempty `<xsl:param>` element has a `select` attribute.

Appears in

`<xsl:stylesheet>` and `<xsl:template>`. If an `<xsl:param>` appears as a child of `<xsl:stylesheet>`, then it is a global parameter visible throughout the stylesheet. XSLT doesn't define the way global parameters are passed to the XSLT processor, so check the documentation for your processor to see how this is done. (See "Global Parameters" in Chapter 5 for an overview of how to pass parameters to the most popular XSLT processors.)

[2.0] In XSLT 2.0, `<xsl:param>` can also appear in the new `<xsl:function>` element.

Defined in

[1.0] XSLT section 11, "Variables and Parameters."

[2.0] XSLT section 9, "Variables and Parameters."

Example

This stylesheet that defines several `<xsl:param>` elements, both global and local. Notice that one of the parameters is a node-set; parameters can be of any XPath or XSLT datatype. Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- param.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>
```

```
  <xsl:param name="favoriteNumber" select="23"/>
  <xsl:param name="favoriteColor"/>


  <xsl:template match="/">
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="list/title"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:variable name="listitems" select="list/listitem"/>
    <xsl:call-template name="processListItems">
      <xsl:with-param name="items" select="$listitems"/>
      <xsl:with-param name="color" select="'yellow'"/>
      <xsl:with-param name="number" select="$favoriteNumber"/>
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="processListItems">
    <xsl:param name="items"/>
    <xsl:param name="color" select="'blue'"/>

    <xsl:for-each select="$items">
      <xsl:value-of select="position()"/>
      <xsl:text>.  </xsl:text>
      <xsl:value-of select="."/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>

    <xsl:text>&#xA;</xsl:text>

    <xsl:text>Your favorite color is </xsl:text>
    <xsl:value-of select="$favoriteColor"/>
    <xsl:text>.</xsl:text>
    <xsl:text>&#xA;</xsl:text>
    <xsl:text>The color passed to this template is </xsl:text>
    <xsl:value-of select="$color"/>
    <xsl:text>.</xsl:text>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

We'll use this stylesheet to transform the following document:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list>
  <title>A few of my favorite albums</title>
  <listitem>A Love Supreme</listitem>
  <listitem>Beat Crazy</listitem>
  <listitem>You Could Have it So Much Better</listitem>
  <listitem>Kind of Blue</listitem>
  <listitem>London Calling</listitem>
  <listitem>Remain in Light</listitem>
  <listitem>The Joshua Tree</listitem>
```

```
  <listitem>The Indestructible Beat of Soweto</listitem>
</list>
```

Here are the results:

```
A few of my favorite albums
1.  A Love Supreme
2.  Beat Crazy
3.  You Could Have it So Much Better
4.  Kind of Blue
5.  London Calling
6.  Remain in Light
7.  The Joshua Tree
8.  The Indestructible Beat of Soweto

Your favorite color is purple.
The color passed to this template is yellow.
```

Notice that when we call the template `processListItems`, we passed in three parameters, only two of which are defined inside the template. In XSLT 1.0, the undefined parameters are simply ignored; in XSLT 2.0, this is a fatal error. If we change the stylesheet to `<xsl:stylesheet version="2.0" ...`, the stylesheet won't run at all. Also notice that the parameter `$items` is a node-set, so we can use the parameter in an `<xsl:for-each>` element.

To generate the previous results, we passed the value `purple` to the XSLT processor. If you're using Saxon, the command line looks like this:

```
java net.sf.saxon.Transform albums.xml param.xsl favoriteColor=purple
```

With Xalan-J, the value is passed like this:

```
java org.apache.xalan.xslt.Process -in albums.xml -xsl param.xsl
  -param favoriteColor purple
```

(The command should be entered on a single line.) See "Global Parameters" in Chapter 5 for a more complete discussion of global parameters and how they can be set for various XSLT processors.

As a further example, here's an XSLT 2.0 stylesheet that uses required parameters, including a required global parameter that must be specified when the stylesheet is invoked:

```
<?xml version="1.0"?>
<!-- param2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:param name="favoriteNumber"
    required="yes" as="xs:integer"/>

  <xsl:template match="/">
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="list/title"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:variable name="listitems" select="list/listitem"/>
```

```
    <xsl:call-template name="processListItems">
      <xsl:with-param name="items" select="$listitems"/>
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="processListItems">
    <xsl:param name="items" required="yes"/>

    <xsl:for-each select="$items">
      <xsl:value-of select="position()"/>
      <xsl:text>.  </xsl:text>
      <xsl:value-of select="."/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>

    <xsl:text>&#xA;</xsl:text>

    <xsl:text>Your favorite number is </xsl:text>
    <xsl:value-of select="$favoriteNumber"/>
    <xsl:text>.</xsl:text>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

This generates results similar to our earlier stylesheet, but the required parameter is enforced. We'll look at four invocations of this stylesheet with Saxon. For the first, we won't supply the required parameter at all:

```
C:\>java net.sf.saxon.Transform albums.xml param2.xsl
Error
  XTDE0050: No value supplied for required parameter favoriteNumber
Transformation failed: Run-time errors were reported
```

Next we'll specify a value for the required parameter, but that value isn't an integer:

```
C:\>java net.sf.saxon.Transform albums.xml param2.xsl favoriteNumber=yellow
Validation error
  FORG0001: Cannot convert string "yellow" to an integer
Transformation failed: Run-time errors were reported
```

Now we'll supply a valid number, but we mistype the variable name. Variables in XSLT are case-sensitive, so `favoritenumber` and `favoriteNumber` are two different variables.

```
C:\>java net.sf.saxon.Transform albums.xml param2.xsl favoritenumber=23
Error
  XTDE0050: No value supplied for required parameter favoriteNumber
Transformation failed: Run-time errors were reported
```

Finally, we provide the required parameter with a valid value, and everything works:

```
C:\>java net.sf.saxon.Transform albums.xml param2.xsl favoriteNumber=23

Albums I've bought recently:
1.  The Sacred Art of Dub
2.  Only the Poor Man Feel It
3.  Excitable Boy
```

```
4.  Aki Special
5.  Combat Rock
6.  Talking Timbuktu
7.  The Birth of the Cool

Your favorite number is 23.
```

[2.0] <xsl:perform-sort>

Sorts a sequence. The sequence to be sorted can be defined with the `select` attribute, or it can be constructed inside the `<xsl:perform-sort>` element itself.

Category

Instruction

Required Attributes

None.

Optional Attribute

`select`

An XPath expression that defines the items to be sorted. If the `select` attribute is present, the `<xsl:perform-sort>` element can only contain `<xsl:sort>` and `<xsl:fallback>` elements. In other words, with a `select` attribute, the `<xsl:perform-sort>` element is not allowed to construct a sequence of items because the `select` attribute has already specified one.

Content

`<xsl:perform-sort>` contains one or more `<xsl:sort>` elements, along with any number of `<xsl:fallback>` elements. If the `select` attribute is not defined, `<xsl:perform-sort>` can contain a sequence constructor. If there is no `select` attribute and the `<xsl:perform-sort>` element doesn't create a sequence, the result is an empty sequence.

Appears in

Any XSLT element whose content model is a sequence constructor or any literal result element.

Defined in

XSLT section 13.2, "Creating a Sorted Sequence."

Example

We'll start with an `<xsl:perform-sort>` element that uses a `select` attribute to produce a sorted sequence. Here is our stylesheet:

```
<?xml version="1.0"?>
<!-- perform-sort4.xsl -->
```

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:variable name="vendorsInOrder" as="xs:string*">
    <xsl:perform-sort select="/report/brand/name">
      <xsl:sort select="."/>
    </xsl:perform-sort>
  </xsl:variable>

  <xsl:template match="/">
    <xsl:text>We sell these brands of chocolate:&#xA;&#xA;</xsl:text>
    <xsl:value-of select="$vendorsInOrder" separator="&#xA;"/>
  </xsl:template>

</xsl:stylesheet>
```

We'll use this stylesheet to process our document of chocolate bar sales:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

Here are our results:

```
We sell these brands of chocolate:

Callebaut
Ghirardelli
Lindt
Perugina
Valrhona
```

The `select` attribute of our `<xsl:perform-sort>` element creates a sequence of all of the `<name>` elements in our input document. The names of those elements are sorted and returned as a sequence, which is then stored in the variable `vendorsInOrder`. Be aware that `<xsl:perform-sort>` returns a sequence of zero or one or more items.

The `<xsl:sort>` elements can be arbitrarily complex. In this example, the sequence we're creating is still based on the brand names, but the sort order is determined by the sales of each brand:

```
<?xml version="1.0"?>
<!-- perform-sort5.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:variable name="vendorsInOrder" as="xs:string*">
    <xsl:perform-sort select="/report/brand/name">
      <xsl:sort select="../units"
        data-type="number" order="descending"/>
    </xsl:perform-sort>
  </xsl:variable>

  <xsl:template match="/">
    <xsl:text>Here are our best-selling brands:&#xA;&#xA;</xsl:text>
    <xsl:value-of select="$vendorsInOrder" separator="&#xA;"/>
  </xsl:template>

</xsl:stylesheet>
```

Now our results are different:

```
Here are our best-selling brands:

Lindt
Valrhona
Ghirardelli
Perugina
Callebaut
```

We're still using the `select` attribute to define the items in our sequence, but we're sorting them with a more complicated key. For an example of an `<xsl:perform-sort>` element that doesn't use the `select` attribute, we'll use `<xsl:sequence>` to select nodes instead:

```
<?xml version="1.0"?>
<!-- perform-sort6.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:variable name="vendorsInOrder" as="xs:string*">
    <xsl:perform-sort>
      <xsl:sort select="../units"
```

```
        data-type="number" order="descending"/>
      <xsl:sequence select="/report/brand/name"/>
    </xsl:perform-sort>
  </xsl:variable>

  <xsl:template match="/">
    <xsl:text>Here are our best-selling brands:&#xA;&#xA;</xsl:text>
    <xsl:value-of select="$vendorsInOrder" separator="&#xA;"/>
  </xsl:template>

</xsl:stylesheet>
```

This gives the same results as before, although the syntax looks strange. Within `<xsl:perform-sort>`, one or more `<xsl:sort>` elements must appear first. Those elements define the sort key for the sequence of items created inside the `<xsl:perform-sort>` element itself. We could generate a more complicated sequence if we want:

```
<?xml version="1.0"?>
<!-- perform-sort7.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:variable name="vendorsInOrder" as="xs:string*">
    <xsl:perform-sort>
      <xsl:sort select="."
        data-type="number" order="descending"/>
      <xsl:sequence select="/report/brand/units"/>
      <xsl:sequence select="('3829', '28852', '18831')"/>
    </xsl:perform-sort>
  </xsl:variable>

  <xsl:template match="/">
    <xsl:text>Here are our sales figures:&#xA;&#xA;</xsl:text>
    <xsl:value-of select="$vendorsInOrder" separator="&#xA;"/>
  </xsl:template>

</xsl:stylesheet>
```

We've added some bogus sales data to make our figures look better. The results here show that we added extra values to the sequence before it was sorted:

```
Here are our sales figures:

28852
27408
22101
19268
18831
14336
8203
3829
```

These are sorted in reverse order because we used the attribute `order="descending"` on the `<xsl:sort>` element.

<xsl:preserve-space>

Defines the source document elements for which whitespace should be preserved.

Category

Top-level element.

Required Attribute

`elements`

Contains a space-separated list of source document elements for which nonsignificant whitespace should be preserved. *Nonsignificant whitespace* typically means text nodes that contain nothing but whitespace; whitespace that appears in and around text is always preserved. The `elements` attribute can also contain the value `*`, which means whitespace should be preserved in all elements not specified in a `<xsl:strip-space>` element. (Although `<xsl:preserve-space elements="*"/>` is legal, there's no need to ever use this. XSLT preserves whitespace text nodes by default.) It is also valid to specify `elements="po:*"` to specify all elements in the `po` namespace.

[2.0] In XSLT 2.0, the `elements` attribute can use wildcards for the namespace prefix. For example, the value `elements="*:title"` refers to all `<title>` elements, regardless of their namespaces. As with XSLT 1.0, the value `elements="dc:*"` refers to all elements in the `dc` namespace.

The XML spec defines the seldom used attribute `xml:space`. The `xml:space` attribute can have the values `preserve` or `default`. If `xml:space="preserve"` applies to a given element in the XML source, all whitespace is preserved, regardless of any `<xsl:preserve-space>` or `<xsl:strip-space>` elements.

Optional Attributes

None.

Content

None. `<xsl:preserve-space>` is an empty element.

Appears in

`<xsl:preserve-space>` is a top-level element and can only appear as a child of `<xsl:stylesheet>`.

Defined in

[1.0] XSLT section 3.4, "Whitespace Stripping."

[2.0] XSLT section 4.4, "Stripping Whitespace from a Source Tree."

Example

We'll illustrate how <preserve-space> works with the following stylesheet:

```
<?xml version="1.0"?>
<!-- preserve-space.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:strip-space elements="*"/>
  <xsl:preserve-space elements="listing"/>

  <xsl:template match="/">
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="/code-sample/title"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:for-each select="/code-sample/listing">
      <xsl:value-of select="."/>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

We'll use this stylesheet to process the following document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- whitespace.xml -->
<code-sample>
  <title>Conditional variable initialization:</title>
  <listing>
    <type>int</type><var> y</var><endstmt>;</endstmt>
    <type>int</type><var> x</var><endstmt>;</endstmt>

    <var>y</var> <op>=</op> <const>23</const><endstmt>;</endstmt>

    <block>
      <keyword>if</keyword> (<var>y</var> <comp>></comp> <const>10</const>)
        <var>x</var> <op>=</op> <const>5</const><endstmt>;</endstmt>
      <keyword>else </keyword>
      <keyword>if</keyword> (<var>y</var> <comp>></comp> <const>5</const>)
        <var>x</var> <op>=</op> <const>3</const><endstmt>;</endstmt>
      <keyword>else </keyword>
        <var>x</var> <op>=</op> <const>1</const><endstmt>;</endstmt>
    </block>
  </listing>
</code-sample>
```

When we process this document with our stylesheet, we get these results:

```
Conditional variable initialization:

    int y;
    int x;

    y = 23;

    if (y>10)
        x=5;else if (y>5)
        x=3;else x=1;
```

The <xsl:strip-space> element strips the whitespace off all the elements in the document, so we use <xsl:preserve-space> to preserve whitespace inside the <listing> element. Notice that the whitespace inside the <block> element was stripped. The <xsl:preserve-space> element is useful when you need to exclude an element from whitespace stripping; by default, no whitespace is stripped from any nodes. Compare this example to the one for the <strip-space> element.

<xsl:processing-instruction>

Creates a processing instruction in the output document.

Category

Instruction.

Required Attribute

`name`
: Defines the name (also known as the target) of this processing instruction.

Optional Attribute

[2.0] `select`
: Defines an expression that creates the data for the processing instruction. If the `select` attribute is used, the <xsl:processing-instruction> must be empty.

Content

An XSLT template. The contents of the template become the data of the processing instruction.

Appears in

<xsl:processing-instruction> appears inside a template.

Defined in

[1.0] XSLT section 7.3, "Creating Processing Instructions."

[2.0] XSLT section 11.6, "Creating Processing Instructions."

Example

We'll demonstrate a stylesheet that adds a processing instruction to an XML document. The processing instruction will associate the stylesheet *docbook.xsl* with this XML document. Here is our stylesheet:

```
<?xml version="1.0"?>
<!-- processing-instruction.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml"/>

  <xsl:template match="/">
    <xsl:processing-instruction name="xml-stylesheet">href="docbook/
html/docbook.xsl" type="text/xsl"</xsl:processing-instruction>
    <xsl:copy-of select="."/>
  </xsl:template>

</xsl:stylesheet>
```

(The `<xsl:processing-instruction>` element should all be on one line.) This stylesheet simply uses the `<xsl:copy-of>` element to copy the input document to the result tree, adding a processing instruction along the way. We'll use our stylesheet with this XML document:

```
<?xml version="1.0"?>
<!-- greeting.xml -->
<greeting>
  Hello, World!
</greeting>
```

When we run this transformation, here are the results:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="docbook/html/docbook.xsl" type="text/xsl"?>
<greeting>
  Hello, World!
</greeting>
```

Note that the contents of a processing instruction are text. Even though the processing instruction we just generated looks like it contains two attributes, you can't create the processing instruction like this:

```
<?xml version="1.0"?>
<!-- processing-instruction-doesnt-work.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml"/>

  <xsl:template match="/">
    <xsl:processing-instruction name="xml-stylesheet">

      <!-- This doesn't work!  You can't put <xsl:attribute>
           elements inside a <xsl:processing-instruction> element. -->

      <xsl:attribute name="href">
        <xsl:text>docbook/html/docbook.xsl</xsl:text>
```

```
      </xsl:attribute>
      <xsl:attribute name="type">
        <xsl:text>text/xsl</xsl:text>
      </xsl:attribute>
    </xsl:processing-instruction>
    <xsl:copy-of select="."/>
  </xsl:template>

</xsl:stylesheet>
```

If you try this, Xalan throws an exception. Saxon doesn't throw an exception, but it doesn't generate a valid processing instruction, either.

[2.0] <xsl:result-document>

Creates a final result tree. Typically a result tree is written to a file, although an XSLT 2.0 processor is not required to be able to do so. The `<xsl:result-document>` instruction is useful for generating multiple files from a single stylesheet. It also makes it possible to generate the name of the output file at runtime.

Category

Instruction.

Required Attributes

None.

Optional Attributes

`format`
: Refers to an output specification defined on a named `<xsl:output>` element. You can define properties such as an output method or a character encoding, then reuse those output specifications by referencing the `name` attribute of the appropriate `<xsl:output>` element.

`href`
: Defines the URI of this result document. Typically this is used to define a filename for the result document, although an XSLT processor is free to use this URI any way it chooses.

[2.0 – Schema] `type`
: Defines the datatype of the document element. The datatype can be any of the built-in datatypes, or it can be a datatype defined in a schema if you have a schema-aware XSLT 2.0 processor.

  The `type` and `validation` attributes are mutually exclusive.

[2.0 – Schema] `validation`
: Defines how the new document element will be validated. The `validation` attribute has four values: `strict`, `lax`, `preserve`, or `strip`.

`validation="strict"` means that the XSLT processor looks in all the declared schemas for an element declaration (`<xs:element>`) with the same name as the document element. It is a fatal error if the processor can't find a matching `<xs:element>`. Assuming the processor finds the declaration of the element, it validates the new document against the document element's declaration.

`validation="lax"` works just like `validation="strict"`, except that no error occurs if the processor can't find the declaration of the element in any of the in-scope schemas. In that case, the type annotation of the document element is `xs:untyped`.

The value `validation="preserve"` means the created element will have a type annotation of `xs:anyType`, and the type annotations of any nodes the new document element contains will be retained without changes. No schema validation is done.

Finally, `validation="strip"` means that the new document element will have a type annotation of `xs:anyType`.

The `validation` and `type` attributes are mutually exclusive.

`method`
: Defines the output method for this result tree. Valid values are `xml`, `html`, `xhtml`, and `text`. XSLT processors are free to support other methods as well; check your processors documentation to see whether it supports any other output methods.

`byte-order-mark`
: Defines whether or not a byte-order mark is written to the output. Valid values are `yes` and `no`. The default is `yes` if the `encoding` is `UTF-16`, but the default for `UTF-8` can vary from one XSLT processor to the next. The default is `no` for every other encoding.

`cdata-section-elements`
: Lists the elements that should be written as `CDATA` sections in the output document. All restrictions and escaping conventions of `CDATA` sections are handled by the XSLT processor. If you need to list more than one element, separate the element names with one or more whitespace characters. This attribute is used only with `method="xml"` and `method="xhtml"`.

`doctype-public`
: Defines the value of the `PUBLIC` attribute of the `DOCTYPE` declaration in the output document. This attribute defines the public identifier of the output document's DTD. This attribute is ignored with `method="text"`.

`doctype-system`
: Defines the value of the `SYSTEM` attribute of the `DOCTYPE` declaration in the output document. It defines the system identifier of the output document's DTD. This attribute is ignored with `method="text"`.

`encoding`
: Defines the value of the `encoding` attribute of the XML declaration in the output document. This attribute is ignored with `method="text"`.

`escape-uri-attributes`
: Defines whether HTML and XHTML attributes with URI values should have special characters replaced with their hex equivalents. For example, if a URI contains spaces, the escaped value of the URI replaces each space with `%20`.

`include-content-type`
: Defines whether the content type should be written to the output document. Valid values are `yes` and `no`; the default is `yes`. As an example, if `include-content-type` is used for `method="html"`, the element `<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">` is added to the `<head>` element of the HTML document.

`indent`
: Defines whether or not elements in the result tree should be indented. Valid values are `yes` and `no`. This attribute is ignored for `method="text"`.

`media-type`
: Defines the MIME type of the output document. The default value is `text/xml` for `method="xml"`, `text/html` for `method="html"` and `method="xhtml"`, and `text/plain` for `method="text"`.

`normalization-form`
: Valid values are `NFC`, `NFD`, `NFKC`, `NFKD`, `fully-normalized`, and `none`. XSLT processors are free to support other values as well; see your processor's documentation to find out whether it supports any other normalization forms.

`omit-xml-declaration`
: Defines whether or not the result tree should include an XML declaration. Valid values are `yes` and `no`; the default is `no`.

`standalone`
: Defines whether or not the XML declaration should include `standalone`. Valid values are `yes` (the declaration includes `standalone="yes"`), `no` (the declaration includes `standalone="no"`), and (*[2.0]*) `omit` (the declaration does not contain `standalone` at all).

`undeclare-prefixes`
: Defines whether the output document should include namespace undeclarations (an undeclaration associates a namespace prefix with an empty string, as in `xmlns:doug=""`). Valid values are `yes` and `no`. See section 20 of the XSLT 2.0 specification for the complete details on this obscure attribute.

`use-character-maps`
: Defines a space-separated list of named character maps that should be used when creating the output document.

`output-version`
: Overrides the `version` attribute on the `<xsl:output>` element.

Content

A sequence constructor.

Appears in

An <xsl:template>.

Defined in

XSLT section 19.1, "Creating Final Result Trees."

Example

We'll illustrate the power of <xsl:result-document> with this small DocBook document:

```
<?xml version="1.0"?>
<!-- chapters.xml -->
<book>
  <title>XSLT Topics</title>
  <chapter>
    <title>XPath</title>
    <para>If this chapter had any text, it would appear here.</para>
  </chapter>
  <chapter>
    <title>Stylesheet Basics</title>
    <para>If this chapter had any text, it would appear here.</para>
  </chapter>
  <chapter>
    <title>Branching and Control Elements</title>
    <para>If this chapter had any text, it would appear here.</para>
  </chapter>
  <chapter>
    <title>Functions</title>
    <para>If this chapter had any text, it would appear here.</para>
  </chapter>
  <chapter>
    <title>Creating Links and Cross-References</title>
    <para>If this chapter had any text, it would appear here.</para>
  </chapter>
  <chapter>
    <title>Sorting and Grouping Elements</title>
    <para>If this chapter had any text, it would appear here.</para>
  </chapter>
  <chapter>
    <title>Combining XML Documents</title>
    <para>If this chapter had any text, it would appear here.</para>
  </chapter>
</book>
```

We'll use a stylesheet to create several documents. First of all, we'll create a single HTML file that lists all of the titles of the chapters defined here. Each chapter title will be a link to a separate HTML file created with <xsl:result-document>. We'll use <xsl:result-document> to create a separate HTML file from each <chapter> element. Here's the stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- result-document.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" include-content-type="no"/>

  <xsl:template match="/">
    <html>
      <head>
        <title><xsl:value-of select="book/title"/></title>
      </head>
      <body>
        <h1><xsl:value-of select="book/title"/></h1>
        <p>Here are some interesting XSLT topics:</p>
        <ul>
          <xsl:for-each select="book/chapter">
            <li>
              <a href="{concat('chapter', position(), '.html')}">
                <xsl:value-of select="title"/>
              </a>
            </li>
          </xsl:for-each>
        </ul>
        <xsl:for-each select="book/chapter">
          <xsl:result-document method="html"
            include-content-type="no"
            href="{concat('chapter', position(), '.html')}">
            <html>
              <head>
                <title><xsl:value-of select="title"/></title>
              </head>
              <body>
                <h1><xsl:value-of select="title"/></h1>
                <xsl:apply-templates select="*[position() > 1]"/>
              </body>
            </html>
          </xsl:result-document>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="para">
    <p><xsl:apply-templates select="*|text()"/></p>
  </xsl:template>

</xsl:stylesheet>
```

Our stylesheet begins by creating the main HTML document. It uses `<xsl:for-each select="book/chapter">` to create a list item and link for each `<chapter>` in the original document. It then uses another `<xsl:for-each>` element that uses `<xsl:result-document>` to create a new document for each `<chapter>`.

Notice that in the stylesheet we use the attribute value template `{concat('chapter', position( ), '.html')}` to create filenames and links. We could have required an identifying attribute on each `<chapter>` element, but it's simpler to use this naming convention to create the files and links. The fourth `<chapter>` is in the file *chapter4.html*. The main HTML file looks like this:

```
<html>
   <head>
      <title>XSLT Topics</title>
   </head>
   <body>
      <h1>XSLT Topics</h1>
      <p>Here are some interesting XSLT topics:</p>
      <ul>
         <li><a href="chapter1.html">XPath</a></li>
         <li><a href="chapter2.html">Stylesheet Basics</a></li>
         <li><a href="chapter3.html">Branching and Control Elements</a></li>
         <li><a href="chapter4.html">Functions</a></li>
         <li><a href="chapter5.html">Creating Links and Cross-References</a></li>
         <li><a href="chapter6.html">Sorting and Grouping Elements</a></li>
         <li><a href="chapter7.html">Combining XML Documents</a></li>
      </ul>
   </body>
</html>
```

Each individual HTML file looks something like this:

```
<html>
   <head>
      <title>XPath</title>
   </head>
   <body>
      <h1>XPath</h1>
      <p>If this chapter had any text, it would appear here.</p>
   </body>
</html>
```

In a browser, the main HTML file appears as in Figure A-15.

Clicking on one of the links takes us to one of the individual documents created by `<xsl:result-document>`, as shown in Figure A-16.

The `<xsl:result-document>` element officially replaces a common extension element from most XSLT 1.0 processors. Performing this task with an XSLT 1.0 processor required calls to a processor-specific extension element that created multiple output files. See the example "The document() Function and Sorting" in Chapter 9 for a complete discussion of the most common XSLT 1.0 mechanisms.

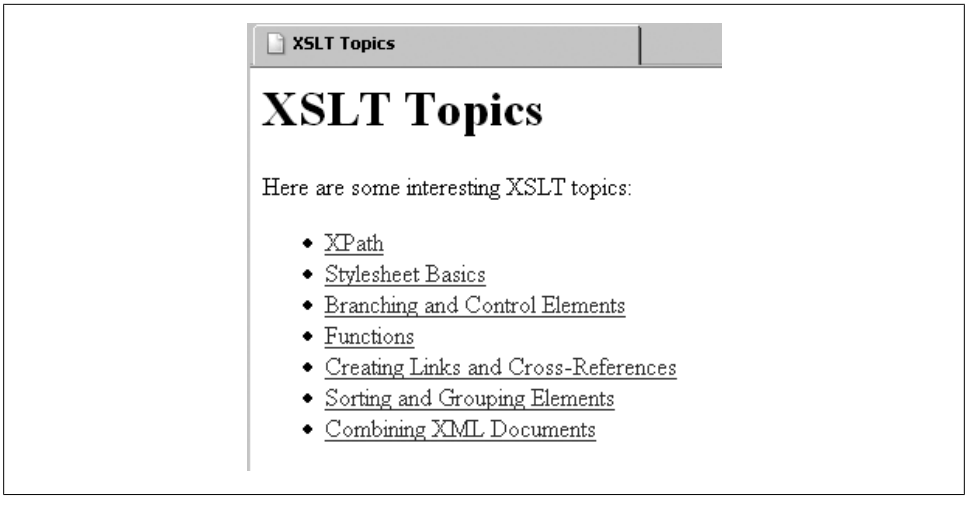


Figure A-15. An HTML file that links to documents created by <xsl:result-document>

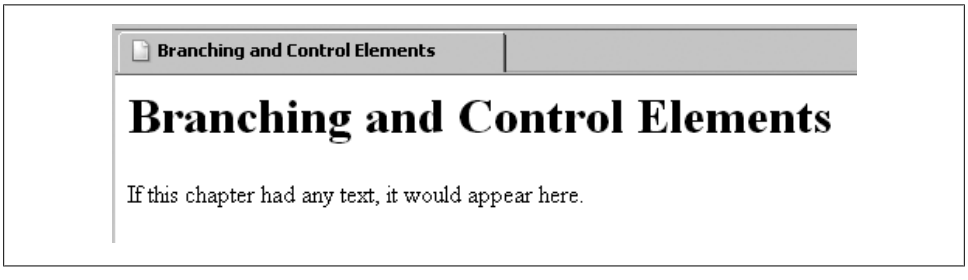


Figure A-16. One of the documents created by <xsl:result-document>

[2.0] <xsl:sequence>

Creates a sequence of nodes and/or atomic values.

Category

Instruction.

Required Attribute

`select`
: An XPath expression that determines the content of the sequence.

Optional Attributes

None.

Content

Zero or more <xsl:fallback> elements. The <xsl:fallback> element specifies processing that should take place in the case of an element that the XSLT processor doesn't support. They are most commonly written for XSLT 1.0 processors operating in forwards-compatible mode.

Appears in

A template.

Defined in

XSLT section 11.10, "Constructing Sequences."

Example

We'll use a simple example here that creates a sequence inside an <xsl:variable>. Here's the stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- sequence.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="sales" as="xs:integer*">
      <xsl:for-each select="/report/brand/units">
        <xsl:if test=". > 10000">
          <xsl:sequence select="."/>
        </xsl:if>
      </xsl:for-each>
    </xsl:variable>

    <xsl:value-of select="/report/title"/>
    <xsl:text>&#xA;&#xA;Sales figures:  </xsl:text>
    <xsl:value-of select="$sales" separator=", "/>
    <xsl:text>&#xA;&#xA;Sequence total:   &#x9;</xsl:text>
    <xsl:value-of select="format-number(sum($sales), '$#,###.00')"/>
    <xsl:text>&#xA;Sequence average:&#x9;</xsl:text>
    <xsl:value-of select="format-number(avg($sales), '$#,###.00')"/>
  </xsl:template>

</xsl:stylesheet>
```

Inside the <xsl:for-each> element, each <xsl:sequence> creates a one-node sequence. The value of the variable is the sequence of xs:integers created from all the items (nodes and atoms) generated by the <xsl:sequence> elements. (In other words, everything selected by a <xsl:sequence> is cast to an xs:integer before it becomes part of the variable.) We'll run this stylesheet against our report of chocolate sales:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
```

```
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

Our stylesheet creates a sequence of all the `<units>` elements larger than 10000. It gives these results:

```
Chocolate bar sales

Sales figures:  27408, 22101, 14336, 19268

Sequence total:         $83,113.00
Sequence average:       $20,778.25
```

Remember that a sequence can have any combination of nodes and atomic values. If we'd like to make our sales figures look better, we could change the declaration of the variable to add more items to the sequence:

```
<!-- sequence2.xsl -->
...
<xsl:variable name="sales" as="xs:integer*">
  <xsl:for-each select="/report/brand/units">
    <xsl:if test=". > 10000">
      <xsl:sequence select="."/>
    </xsl:if>
  </xsl:for-each>
  <xsl:sequence select="(80000, 75000, 65000)"/>
</xsl:variable>
...
```

Our calculations now include the extra numbers, so our sales figures look much better:

```
Chocolate bar sales

Sales figures:  27408, 22101, 14336, 19268, 80000, 75000, 65000

Sequence total:         $303,113.00
Sequence average:       $43,301.86
```

Remember, the variable that contains the sequence is of type `xs:integer*`, so all of the selected nodes and atomic values are cast to `xs:integer`s before they become part of the variable.

<xsl:sort>

Defines a sort key for the current context. The first `<xsl:sort>` defines the primary sort key, the second `<xsl:sort>` defines the secondary sort key, etc. You can have as many `<xsl:sort>` elements as you need.

Category

Subinstruction.

Required Attributes

None.

Optional Attributes

`select`
: Determines the nodes to be sorted. You can also omit the `select` attribute and use the contents of the `<xsl:sort>` element to define the nodes to be sorted.

`lang`
: A string that defines the language used by the sort. The language codes are defined in RFC1766, available at *http://www.ietf.org/rfc/rfc1766.txt*. Not all processors support all languages, so check your XSLT processor's documentation.

`data-type`
: An attribute that defines the type of the items to be sorted. Allowable values are `number` and `text`; the default is `text`. An XSLT processor has the option of supporting other values as well. Sorting the values `32 10 120` with `data-type="text"` returns `10 120 32`, while `data-type="number"` returns `10 32 120`.

  [2.0] In XSLT 2.0, you can specify a QName that represents some other datatype. How (or if) that value is processed can vary from one XSLT processor to another.

`order`
: An attribute that defines the order of the sort. Allowable values are `ascending` and `descending`.

`case-order`
: An attribute that defines the order in which upper- and lowercase letters are sorted. Allowable values are `upper-first` and `lower-first`.

[2.0] `collation`
: The collation order used by this sort. For example, in Spanish, the single letter *ll* sorts after the two letters *lz*, so English collating rules don't sort Spanish words correctly.

This value is a URI; how that URI is used to specify a particular language value varies from one XSLT processor to the next. Keep in mind that if an XSLT 2.0 processor does not support the collation order you requested, the processor uses its default collation.

[2.0] `stable`
: Valid values are `yes` and `no`; the default is `yes`. Only the first `<xsl:sort>` element can have a `stable` attribute. If `stable="yes"`, any elements with the same sort value stay in their original order in the document. For example, if we sort these elements:

```
<statelist>
  <title>U.S. Road trips I've made recently</title>
  <state year="2006" month="4">California</state>
  <state year="2005" month="12">Texas</state>
  <state year="2006" month="2">Washington</state>
  <state year="2006" month="3">California</state>
</statelist>
```

and if we sort by the `year` attribute and the value of the `<state>` element (the `month` attribute is ignored), the first and last elements here have the same sort value. With `stable="yes"`, the two trips to California will stay in their original document order. If you go out of your way to code `stable="no"`, the XSLT processor is not required to keep the nodes in their original order.

Content

[1.0] None.

[2.0] In XSLT 2.0, you can use the content of the `<xsl:sort>` element to define the nodes to be sorted. As you would expect, it is a fatal error to have both a `select` attribute and content inside `<xsl:sort>`. If neither is specified, it defaults to the current context node, which is equivalent to `select="."`

Appears in

`<xsl:apply-templates>` and `<xsl:for-each>`.

[2.0] In XSLT 2.0, `<xsl:sort>` can appear in the new `<xsl:for-each-group>` and `<xsl:perform-sort>` elements as well.

Defined in

[1.0] XSLT section 10, "Sorting."

[2.0] XSLT section 13, "Sorting."

Example

We'll illustrate `<xsl:sort>` with this stylesheet:

```
<?xml version="1.0"?>
<!-- sort.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
<xsl:output method="text"/>

<xsl:template match="/">
  <xsl:text>&#xA;</xsl:text>
  <xsl:call-template name="ascending-alpha-sort">
    <xsl:with-param name="items" select="/sample/textlist/listitem"/>
  </xsl:call-template>
  <xsl:call-template name="ascending-alpha-sort">
    <xsl:with-param name="items" select="/sample/numericlist/listitem"/>
  </xsl:call-template>
  <xsl:call-template name="ascending-numeric-sort">
    <xsl:with-param name="items" select="/sample/numericlist/listitem"/>
  </xsl:call-template>
  <xsl:call-template name="descending-alpha-sort">
    <xsl:with-param name="items" select="/sample/textlist/listitem"/>
  </xsl:call-template>
  <xsl:call-template name="category-sort">
    <xsl:with-param name="items" select="/sample/textlist/listitem"/>
  </xsl:call-template>
</xsl:template>

<xsl:template name="ascending-alpha-sort">
  <xsl:param name="items"/>
  <xsl:text>Ascending text sort:</xsl:text>
  <xsl:text>&#xA;</xsl:text>
  <xsl:for-each select="$items">
    <xsl:sort select="."/>
    <xsl:value-of select="."/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:for-each>
  <xsl:text>&#xA;</xsl:text>
</xsl:template>

<xsl:template name="descending-alpha-sort">
  <xsl:param name="items"/>
  <xsl:text>Descending text sort:</xsl:text>
  <xsl:text>&#xA;</xsl:text>
  <xsl:for-each select="$items">
    <xsl:sort select="." order="descending"/>
    <xsl:value-of select="."/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:for-each>
  <xsl:text>&#xA;</xsl:text>
</xsl:template>

<xsl:template name="ascending-numeric-sort">
  <xsl:param name="items"/>
  <xsl:text>Ascending numeric sort:</xsl:text>
  <xsl:text>&#xA;</xsl:text>
  <xsl:for-each select="$items">
    <xsl:sort select="." data-type="number"/>
    <xsl:value-of select="."/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:for-each>
```

```
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

  <xsl:template name="category-sort">
    <xsl:param name="items"/>
    <xsl:text>Ascending category sort:</xsl:text>
    <xsl:text>&#xA;</xsl:text>
    <xsl:for-each select="$items">
      <xsl:sort select="@category"/>
      <xsl:sort select="."/>
      <xsl:value-of select="."/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

Our stylesheet defines four named templates, each of which sorts `<listitem>`s in a different order or with a different `data-type`. Notice that the fourth template, `category-sort`, sorts by an attribute first, then by the value of the current item. We'll use this stylesheet against this document:

```
<?xml version="1.0"?>
<!-- items-to-sort.xml -->
<sample>
  <numericlist>
    <listitem>1</listitem>
    <listitem>3</listitem>
    <listitem>23</listitem>
    <listitem>120</listitem>
    <listitem>2</listitem>
  </numericlist>
  <textlist>
    <listitem category="number">3</listitem>
    <listitem category="fruit">apple</listitem>
    <listitem category="fruit">orange</listitem>
    <listitem category="foreign-fruit">dragonfruit</listitem>
    <listitem category="foreign-fruit">carambola</listitem>
  </textlist>
  <datelist>
    <listitem>2006-12-25</listitem>
    <listitem>1995-04-21</listitem>
    <listitem>1965-06-19</listitem>
    <listitem>2007-01-01</listitem>
  </datelist>
  <spanishlist>
    <listitem>campo</listitem>
    <listitem>luna</listitem>
    <listitem>ciudad</listitem>
    <listitem>llaves</listitem>
    <listitem>chihuahua</listitem>
    <listitem>arroz</listitem>
    <listitem>limonada</listitem>
```

```
  </spanishlist>
</sample>
```

Here are the results:

```
Ascending text sort:
3
apple
carambola
dragonfruit
orange

Ascending text sort:
1
120
2
23
3

Ascending numeric sort:
1
2
3
23
120

Descending text sort:
orange
dragonfruit
carambola
apple
3

Ascending category sort:
carambola
dragonfruit
apple
orange
3
```

Notice that the `data-type="numeric"` attribute causes data to be sorted in numeric order.

[2.0] To illustrate some of the new features of XSLT 2.0, we'll sort this data with another stylesheet. One template casts the children of `<datelist>` into `xs:date` values, and then sorts those values. Another template uses a custom collation to sort the children of `<spanishlist>` using a custom collation. Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- sort2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
```

```
  <xsl:text>&#xA;</xsl:text>
  <xsl:variable name="date-sequence" as="xs:date*">
    <xsl:for-each select="/sample/datelist/listitem">
      <xsl:value-of select="xs:date(.)"/>
    </xsl:for-each>
  </xsl:variable>
  <xsl:call-template name="ascending-date-sort">
    <xsl:with-param name="items" select="$date-sequence"/>
  </xsl:call-template>
  <xsl:call-template name="spanish-alpha-sort">
    <xsl:with-param name="items" select="/sample/spanishlist/listitem"/>
  </xsl:call-template>
  <xsl:call-template name="ascending-alpha-sort">
    <xsl:with-param name="items" select="/sample/spanishlist/listitem"/>
  </xsl:call-template>
</xsl:template>

<xsl:template name="ascending-date-sort">
  <xsl:param name="items" as="xs:date*"/>
  <xsl:text>Ascending date sort:</xsl:text>
  <xsl:text>&#xA;</xsl:text>
  <xsl:for-each select="$items">
    <xsl:sort/>
    <xsl:value-of select="."/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:for-each>
  <xsl:text>&#xA;</xsl:text>
</xsl:template>

<xsl:template name="ascending-alpha-sort">
  <xsl:param name="items"/>
  <xsl:text>Ascending text sort:</xsl:text>
  <xsl:text>&#xA;</xsl:text>
  <xsl:for-each select="$items">
    <xsl:sort/>
    <xsl:value-of select="."/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:for-each>
  <xsl:text>&#xA;</xsl:text>
</xsl:template>

<xsl:template name="spanish-alpha-sort">
  <xsl:param name="items"/>
  <xsl:text>Spanish alpha sort:</xsl:text>
  <xsl:text>&#xA;</xsl:text>
  <xsl:for-each select="$items">
    <xsl:sort
      collation="{concat('http://saxon.sf.net/collation?',
                  'class=com.oreilly.xslt.SpanishCollation;')}"/>
    <xsl:value-of select="."/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:for-each>
  <xsl:text>&#xA;</xsl:text>
</xsl:template>
```

```
</xsl:stylesheet>
```

The results look like this:

```
Ascending date sort:
1965-06-19
1995-04-21
2006-12-25
2007-01-01

Spanish alpha sort:
arroz
campo
ciudad
chihuahua
limonada
luna
llaves

Ascending text sort:
arroz
campo
chihuahua
ciudad
limonada
llaves
luna
```

The dates are sorted as `xs:date` values, even though those results are the same as a text sort. For the Spanish collation, we specify the Java class that compares strings to determine the sort order. The way the Java class is specified here is specific to the Saxon XSLT 2.0 processor. Check the documentation of your XSLT processor to see how (or if) it supports custom collations.

See Chapter 9 for a discussion of "Creating Custom Collations."

<xsl:strip-space>

Defines the source-document elements for which nonsignificant whitespace should be removed.

Category

Top-level element.

Required Attribute

`elements`

Contains a space-separated list of element names or node tests for which nonsignificant whitespace should be removed. *Nonsignificant whitespace* typically means text nodes that contain nothing but whitespace; whitespace that appears in and around text is always preserved. The `elements` attribute can also contain the

value *, which means whitespace should be removed from all elements not specified in an <xsl:preserve-space> element. The value elements="auth:*" is also legal; this specifies all elements in the auth namespace.

[2.0] In XSLT 2.0, the elements attribute can also use wildcards for the namespace prefix. For example, the value *:title refers to all <title> elements, regardless of their namespaces.

The XML spec defines the seldom used attribute xml:space. The xml:space attribute can have the values preserve or default. If xml:space="preserve" applies to a given element in the XML source, all whitespace is preserved, regardless of any <xsl:preserve-space> or <xsl:strip-space> elements.

Optional Attributes

None.

Content

None. <xsl:strip-space> is an empty element.

Appears in

<xsl:strip-space> is a top-level element and can only appear as a child of <xsl:stylesheet>.

Defined in

[1.0] XSLT section 3.4, "Whitespace Stripping."

[2.0] XSLT section 4.4, "Stripping Whitespace from a Source Tree."

Example

We'll illustrate the <xsl:strip-space> element with the following stylesheet:

```
<?xml version="1.0"?>
<!-- strip-space.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:strip-space elements="*"/>

  <xsl:template match="/">
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="/code-sample/title"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:for-each select="/code-sample/listing">
      <xsl:value-of select="."/>
    </xsl:for-each>
  </xsl:template>
```

```
</xsl:stylesheet>
```

We'll use this stylesheet to process the following document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- whitespace.xml -->
<code-sample>
  <title>Conditional variable initialization:</title>
  <listing>
    <type>int</type><var> y</var><endstmt>;</endstmt>
    <type>int</type><var> x</var><endstmt>;</endstmt>

    <var>y</var> <op>=</op> <const>23</const><endstmt>;</endstmt>

    <block>
      <keyword>if</keyword> (<var>y</var> <comp>></comp> <const>10</const>)
        <var>x</var> <op>=</op> <const>5</const><endstmt>;</endstmt>
      <keyword>else </keyword>
      <keyword>if</keyword> (<var>y</var> <comp>></comp> <const>5</const>)
        <var>x</var> <op>=</op> <const>3</const><endstmt>;</endstmt>
      <keyword>else </keyword>
        <var>x</var> <op>=</op> <const>1</const><endstmt>;</endstmt>
    </block>
  </listing>
</code-sample>
```

Here are the results:

```
Conditional variable initialization:
int y;int x;y=23;if (y>10)
        x=5;else if (y>5)
        x=3;else x=1;
```

Notice that all the extra whitespace from all the elements has been removed. This includes the space between the various elements contained inside `<listing>`, such as `<keyword>`, `<const>`, and `<var>`. Wherever the `<listing>` element has nonwhitespace characters between elements (a semicolon or parenthesis, for example), that text node and all the whitespace it contains are preserved. Those text nodes are the reason why some whitespace appears in our results. If we went to the extreme of putting every character inside an element, all of the text inside the `<listing>` element would run together.

Compare this to the example for `<xsl:preserve-space>`.

<xsl:stylesheet>

The root element of an XSLT stylesheet. It is identical to the `<xsl:transform>` element, which was included in the XSLT specification for historical purposes.

Category

Contains the entire stylesheet.

Required Attributes

`version`
: Indicates the version of XSLT that the stylesheet requires. For XSLT version 1.0, its value should always be `1.0`. As later versions of the XSLT specification are defined, the required values for the `version` attribute will be defined along with them.

  [2.0] In XSLT version 2.0, this value should be `2.0`. The XSLT 2.0 spec defines stricter rules for handling the `version` attribute than the XSLT 1.0 spec. To avoid complications, you should use the values `1.0` and `2.0`.

  XSLT 2.0 also allows you to specify the `version` attribute on any XSLT element. If you want to use version 1.0 processing for a given portion of your stylesheet, you can define `version="1.0"` for that portion of the stylesheet. For example, the `<xsl:decimal-format>` element works differently in XSLT 2.0, so you could use `<xsl:decimal-format version="1.0">` to process the element according to the rules of XSLT 1.0.

`xmlns:xsl`
: Defines the URI for the XSL namespace. For both XSLT version 1.0 and XSLT version 2.0, this attribute's value must be `http://www.w3.org/1999/XSL/Transform`. (Although this is a namespace declaration, not an attribute, it is required. Your stylesheet won't work without it.)

Optional Attributes

`id`
: Defines an ID for this stylesheet.

`extension-element-prefixes`
: Defines any namespace prefixes used to invoke extension elements. Multiple namespace prefixes are separated by whitespace.

`exclude-result-prefixes`
: Defines namespace prefixes whose declarations should not be sent to the output document. Multiple namespace prefixes are separated by whitespace.

[2.0] `xpath-default-namespace`
: Defines the default namespace used by XPath functions and operators. If you're transforming a document that uses a default namespace (`http://www.oreilly.com`, for example), you must normally namespace-qualify the names of any elements in your XPath expressions. Defining `xpath-default-namespace="http://www.oreilly.com"` tells XPath to use this namespace as the default. By using this on any element in your stylesheet, you can have different default XPath namespaces in different sections of your stylesheet.

[2.0] `default-validation`
: Defines the default value for the `validation` attribute of the `<xsl:document>`, `<xsl:element>`, `<xsl:attribute>`, `<xsl:copy>`, `<xsl:copy-of>`, and `<xsl:result-`

document> elements. Allowed values for this attribute are preserve and strip. If this attribute is not coded, the default value is strip.

[2.0] default-collation
: A series of space-separated URIs that define the default collation sequence. This sequence is used by <xsl:key> and <xsl:for-each-group>, but it does not affect the collation used by <xsl:sort>. The way collation sequences are defined varies from one XSLT processor to another, so check your processor's documentation for information on defining a collation sequence.

[2.0] input-type-annotations
: Defines whether datatype information should be preserved in the output. Valid values for this attribute are preserve, strip, and unspecified; the default value is unspecified.

Content

This element contains the entire stylesheet. The following elements can be children of <xsl:stylesheet>:

<xsl:attribute-set>
[2.0] [2.0] <xsl:character-map>
<xsl:decimal-format>
[2.0] [2.0] <xsl:function>
<xsl:import>
[2.0] [2.0 – Schema] <xsl:import-schema>
<xsl:include>
<xsl:key>
<xsl:namespace-alias>
<xsl:output>
<xsl:param>
<xsl:preserve-space>
<xsl:strip-space>
<xsl:template>
<xsl:variable>

Appears in

None. <xsl:stylesheet> is the root element of the stylesheet.

Defined in

[1.0] XSLT section 2.2, "Stylesheet Element."

[2.0] XSLT section 3.6, "Stylesheet Element."

Example

For a simple example, we'll use the Hello World document from the XML 1.0 specification:

```
<?xml version="1.0"?>
<greeting>
  Hello, World!
</greeting>
```

We'll transform our document with this stylesheet:

```
<?xml version="1.0"?>
<!-- stylesheet.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <xsl:apply-templates select="greeting"/>
  </xsl:template>

  <xsl:template match="greeting">
    <html>
      <body>
        <h1>
          <xsl:value-of select="."/>
        </h1>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

When we transform our document with this stylesheet, here are the results:

```
<html>
<body>
<h1>
  Hello, World!
</h1>
</body>
</html>
```

[2.0] XSLT 2.0 added the `xpath-default-namespace` attribute to simplify working with documents that have a default namespace. Here's a modified version of our chocolate sales document that uses a default namespace:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate-default-namespace.xml -->
<report month="8" year="2006"
  xmlns="http://www.oreilly.com">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
```

```
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

Here's a simple stylesheet to list the sales figures for each brand:

```
<?xml version="1.0"?>
<!-- stylesheet2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Here are this month's sales figures:&#xA;&#xA;</xsl:text>
    <xsl:for-each select="/report/brand">
      <xsl:value-of select="name"/>
      <xsl:text>&#x9;</xsl:text>
      <xsl:value-of select="units"/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

And here are our disappointing results:

```
Here are this month's sales figures:
```

We don't get any results because we don't specify the namespace of our element names in our XPath expressions. If we add `xpath-default-namespace` to the `<xsl:stylesheet>` element:

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xpath-default-namespace="http://www.oreilly.com">
```

we get the results we want:

```
Here are this month's sales figures:

Lindt   27408
Callebaut       8203
Valrhona        22101
Perugina        14336
Ghirardelli     19268
```

<xsl:template>

Defines an output template. Templates that begin `<xsl:template match="x"` define a transformation for a given element. Templates that begin `<xsl:template name="x"` define a set of output elements that are processed whenever the template is invoked by name. All `<xsl:template>` elements must have either the `match` or the `name` attribute defined. Although not common, it is also possible to create `<xsl:template>` elements that have both a `match` and a `name`.

Category

Top-level element.

Required Attributes

None; however, an `<xsl:template>` must contain a `match` attribute, a `name` attribute, or both.

Optional Attributes

`match`
: A pattern that defines the elements for which this template should be invoked. For example, `<xsl:template match="xyz">` defines a template for processing `<xyz>` elements.

`name`
: An attribute that names this template. Named templates are invoked with the `<xsl:call-template>` element.

`mode`
: An attribute that defines a mode for this template. A *mode* is a convenient syntax that allows you to write specific templates for specific purposes. For example, we could write an `<xsl:template>` with `mode="toc"` to process a node for the table of contents of a document, and we could write other `<xsl:template>`s with `mode="print"`, `mode="online"`, `mode="index"`, etc. to process the same information for different purposes.

  [2.0] In XSLT 2.0, the `mode` attribute can use the value `#default` to indicate that this template applies to the default mode, or use `#all` to indicate that this template applies to all modes. XSLT 2.0 also allows you to specify multiple mode names separated by whitespace. It is an error if you use the `mode` attribute on a template that doesn't have a `match` attribute.

`priority`
: An attribute that assigns a numeric priority to this template. The value can be any numeric value except `Infinity`. If the XSLT processor cannot determine which template to use (in other words, more than one template has the same default priority), the `priority` attribute allows you to define a tiebreaker. Keep in mind that you can simply use the `priority` attribute to raise the priority of a given template.

[2.0] In XSLT 2.0, it is an error if you use the `priority` attribute on a named template.

[2.0] `as`
: Defines the datatype of the result of the template. If this attribute is omitted, the template can return any datatype.

Content

An XSLT template.

Appears in

`<xsl:template>` is a top-level element and can only appear as a child of `<xsl:stylesheet>`.

Defined in

[1.0] XSLT section 5.3, "Defining Template Rules."

[2.0] XSLT section 6.1, "Defining Template Rules."

Example

We'll use the Hello World document from the XML 1.0 specification for our example:

```
<?xml version="1.0"?>
<greeting>
  Hello, World!
</greeting>
```

We'll transform our document with this stylesheet:

```
<?xml version="1.0"?>
<!-- stylesheet.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <xsl:apply-templates select="greeting"/>
  </xsl:template>

  <xsl:template match="greeting">
    <html>
      <body>
        <h1>
          <xsl:value-of select="."/>
        </h1>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

This simple stylesheet has two templates, one of which matches the root node (or document node in XSLT 2.0), and one of which matches the `<greeting>` element (`match="greeting"`).

```
<html>
<body>
<h1>
  Hello, World!
</h1>
</body>
</html>
```

<xsl:text>

Allows you to write literal text to the output document. The main benefit of the `<xsl:text>` element is that it gives you complete control over whitespace in the output.

Category

Instruction.

Required Attributes

None.

Optional Attribute

`disable-output-escaping`

Defines whether special characters are escaped when they are written to the output document. For example, if the literal text contains the character `>`, it is normally written to the output document as `>`. If you code `disable-output-escaping="yes"`, the character `>` is written instead. Note that If you're using `<xsl:output method="text">`, this attribute is ignored because output escaping is not done for the `text` output method.

[2.0] In XSLT 2.0, this attribute is deprecated. Instead you should use the new [2.0] `<xsl:character-map>` element.

Content

Literal text and entity references (`
`, for example). These are known collectively as PCDATA, or *parsed character data*.

Appears in

`<xsl:text>` appears inside a template.

Defined in

[1.0] XSLT section 7.2, "Creating Text."

[2.0] XSLT section 11.4.2, "Creating Text Nodes Using `xsl:text`."

Example

This sample stylesheet generates text with <xsl:text>. We intermingle <xsl:text> and <xsl:value-of> elements to create a coherent sentence. In this case, we simply generate a text document, but this technique works equally well to create the text of an HTML or XML element. Here is the stylesheet:

```
<?xml version="1.0"?>
<!-- text.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Test of &lt;xsl:text&gt;</title>
      </head>
      <body style="font-family: sans-serif;">
        <!-- disable-output-escaping="no" by default -->
        <h1>Test of &lt;xsl:text&gt;</h1>
        <p>
          <xsl:text>Your document contains </xsl:text>
          <xsl:value-of select="count(//*)"/>
          <xsl:text> elements and </xsl:text>
          <xsl:value-of select="count(//@*)"/>
          <xsl:text> attributes.  </xsl:text>
        </p>
        <p>
          <xsl:text
            disable-output-escaping="yes">&lt;Have a great day!&gt;</xsl:text>
        </p>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

Given this XML document:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

Our stylesheet produces this HTML document:

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Test of &lt;xsl:text&gt;</title>
   </head>
   <body style="font-family: sans-serif;">
      <h1>Test of &lt;xsl:text&gt;</h1>
      <p>Your document contains 9 elements and 5 attributes.  </p>
      <p><Have a great day!></p>
   </body>
</html>
```

If you view this document in a browser, the browser ignores the unknown `<Have>` element in the second paragraph. Using the `disable-output-escaping` attribute lets you generate text that is not valid (or even well-formed) HTML, XML, or XHTML, and it should be used as a last resort in an XSLT 1.0 stylesheet.

<xsl:transform>

This is a synonym for `<xsl:stylesheet>`. It was included in the XSLT 1.0 spec for historical purposes. Its attributes, content, and all other properties are the same as those for `<xsl:stylesheet>`. See the "`<xsl:stylesheet>`" section for more information.

Category

Instruction.

<xsl:value-of>

Calculates the value of an XPath expression, converts that value to a text node and then writes the value to the result tree.

Category

Instruction.

Required Attribute

`select`
: The XPath expression that is evaluated and written to the output.

  [2.0] In XSLT 2.0, this attribute is optional. The `<xsl:value-of>` element must have either a `select` attribute or it must contain content. It is an error if it contains neither or both.

Optional Attributes

`disable-output-escaping`
: An attribute that defines whether special characters are escaped when written to the output document. For example, if the literal text contains the character `>`, it is

normally written to the output document as `>`. If you code `disable-output-escaping="yes"`, the character > is written instead. The XSLT processor uses this attribute only if you use the `html` or `xml` output methods. If you use `<xsl:output method="text">`, the attribute is ignored becasue output escaping is not done for the `text` output method. See the "`<xsl:text>`" section for a more thorough discussion of the `disable-output-escaping` attribute.

[2.0] In XSLT 2.0, this attribute is deprecated. You should use a character map instead; see the discussion of the [2.0] `<xsl:character-map>` element for more information.

[2.0] `separator`

Defines the characters that separate multiple values generated by the `select` attribute. The default value is a single space (`#x20`). The value of the `separator` attribute appears after all of the values except the last.

Be aware that in XSLT 1.0, `<xsl:value-of>` selects the first item in a node-set and discards all the others. In XSLT 2.0, all of the nodes are output with the separator between them.

Content

None. `<xsl:value-of>` is an empty element.

[2.0] In XSLT 2.0, `<xsl:value-of>` can contain content; that content is evaluated and written to the output.

Appears in

`<xsl:value-of>` appears inside a template.

Defined in

[1.0] XSLT section 7.6.1, "Generating Text with `xsl:value-of`."

[2.0] XSLT section 11.4.3, "Generating Text with `xsl:value-of`."

Example

We'll use the `<xsl:value-of>` element to generate some text. Here is our stylesheet:

```
<?xml version="1.0"?>
<!-- value-of.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Your document contains </xsl:text>
    <xsl:value-of select="count(//*)"/>
    <xsl:text> elements and </xsl:text>
```

```
    <xsl:value-of select="count(//@*)"/>
    <xsl:text> attributes.&#xA;Have a great day!</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

We'll use this XML document as input:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

Here are the results:

```
Your document contains 9 elements and 5 attributes.
Have a great day!
```

[2.0] Before we leave the `<xsl:value-of>` element, we'll look at its new capabilities. Here's a short stylesheet that uses a nonempty `<xsl:value-of>` and the `separator` attribute:

```
<?xml version="1.0"?>
<!-- value-of2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:value-of>
      <xsl:text>Here is a list of the </xsl:text>
      <xsl:value-of select="count(//*)"/>
      <xsl:text> elements in your document:&#xA;&#xA;</xsl:text>
    </xsl:value-of>
    <xsl:value-of select="//*/name()" separator="&#xA;"/>
  </xsl:template>

</xsl:stylesheet>
```

Processing *albums.xml* with our new stylesheet generates these results:

```
Here is a list of the 9 elements in your document:

list
title
listitem
listitem
listitem
listitem
listitem
```

```
listitem
listitem
```

The first paragraph is generated by an `<xsl:value-of>` element that contains three elements, one of which is another `<xsl:value-of>` element. The rest of the document is generated by a single `<xsl:value-of>` statement that uses the handy `separator` attribute. If we wanted to do the same thing in XSLT 1.0, we would have to replace the following line:

```
<xsl:value-of select="//*/name()" separator="&#xA;"/>
```

with this more complicated markup:

```
<xsl:for-each select="//*">
  <xsl:value-of select="name()"/>
  <xsl:if test="position() != last()">
    <xsl:text>&#xA;</xsl:text>
  </xsl:if>
</xsl:for-each>
```

<xsl:variable>

Defines a variable. If `<xsl:variable>` occurs as a top-level element, it is a global variable that is accessible throughout the stylesheet. Otherwise, the variable is local and exists only in the element that contains the `<xsl:variable>`. The value of the variable can be defined in one of two ways: specified in the `select` attribute or defined in an XSLT template inside the `<xsl:variable>` element itself. If neither method is used, the value of the variable is an empty string.

Category

Either a top-level element or an instruction.

Required Attribute

`name`
: An attribute that names this variable.

Optional Attributes

`select`
: An XPath expression that defines the value of this variable.

[2.0 – Schema] `as`
: The datatype of the variable. For example, `<xsl:variable name="age" as="xs:integer" select="11"/>` defines a new variable of type `xs:integer`. It is an error if the supplied value can't be converted to the specified type; using the attribute `select="'really, really old'"` here would cause an error. If you're using a schema-aware XSLT processor, you can use your own datatypes.

Content

The <xsl:variable> element can be empty or it can contain an XSLT template. It is a fatal error if it contains both content and a select attribute.

Appears in

<xsl:stylesheet> as a top-level element or in a template.

Defined in

[1.0] XSLT section 11, "Variables and Parameters."

[2.0] XSLT section 9, "Variables and Parameters."

Example

Here is a stylesheet that defines a number of variables:

```
<?xml version="1.0"?>
<!-- variable.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:variable name="favoriteNumber" select="23"/>
  <xsl:variable name="favoriteColor" select="'blue'"/>
  <xsl:variable name="complicatedVariable">
    <xsl:choose>
      <xsl:when test="count(//listitem) > 10">
        <xsl:text>really long list</xsl:text>
      </xsl:when>
      <xsl:when test="count(//listitem) > 5">
        <xsl:text>moderately long list</xsl:text>
      </xsl:when>
      <xsl:otherwise>
        <xsl:text>fairly short list</xsl:text>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:variable>

  <xsl:template match="/">
    <xsl:text>Hello!  Your favorite number is </xsl:text>
    <xsl:value-of select="$favoriteNumber"/>
    <xsl:text>.&#xA;Your favorite color is </xsl:text>
    <xsl:value-of select="$favoriteColor"/>
    <xsl:text>.&#xA;&#xA;Here is a </xsl:text>
    <xsl:value-of select="$complicatedVariable"/>
    <xsl:text>:&#xA;</xsl:text>
    <xsl:variable name="listitems" select="list/listitem"/>
    <xsl:call-template name="processListItems">
      <xsl:with-param name="items" select="$listitems"/>
    </xsl:call-template>
  </xsl:template>
```

```
  <xsl:template name="processListItems">
    <xsl:param name="items"/>
    <xsl:variable name="favoriteColor">
      <xsl:text>chartreuse</xsl:text>
    </xsl:variable>

    <xsl:text>    (Your favorite color is now </xsl:text>
    <xsl:value-of select="$favoriteColor"/>
    <xsl:text>.)&#xA;</xsl:text>
    <xsl:for-each select="$items">
      <xsl:value-of select="position()"/>
      <xsl:text>.  </xsl:text>
      <xsl:value-of select="."/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

We'll use our stylesheet to transform the following document:

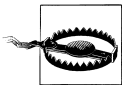
```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

Here are the results of our transformation:

```
Hello!  Your favorite number is 23.
Your favorite color is blue.

Here is a moderately long list:
    (Your favorite color is now chartreuse.)
1.  The Sacred Art of Dub
2.  Only the Poor Man Feel It
3.  Excitable Boy
4.  Aki Special
5.  Combat Rock
6.  Talking Timbuktu
7.  The Birth of the Cool
```

In XSLT 1.0, having two variables at the same level with the same name is an error. It's also an error to define an `<xsl:variable>` and an `<xsl:param>` with the same name at the same level. Neither of these conditions is an error in XSLT 2.0.

Several things are worth mentioning in our stylesheet. First, notice that when we defined values for the first two variables (`favoriteNumber` and `favoriteColor`), we had to quote the string `blue`, but didn't have to quote `23`. If we don't quote `blue`, the XSLT processor assumes we mean the node-set (or sequence) of all the `<blue>` elements in the context node. That's obviously not what we want.



Be aware that single quotes around numeric values are significant. The value `35` represents a numeric value (it's a number in XSLT 1.0 and an `xs:integer` in XSLT 2.0), while the value `'35'` represents the *string* `35`. That might seem like a minor distinction, but it has a major impact on how your stylesheet works, especially in XSLT 2.0.

Also notice that we have two variables named `favoriteColor`. One is a global variable because its parent is the `<xsl:stylesheet>` element; the other is a local variable because it is defined in a `<xsl:template>`. When we access `favoriteColor` in the `match="/"` template, it has one value; when we access it inside the `name="processListItems"` template, it has another.

Using an `<xsl:choose>` element to initialize an `<xsl:variable>` is a common technique. This technique is the equivalent of this procedural programming construct:

```
String complicatedVariable;

if (count(listitems) > 10)
  complicatedVariable = "really long list";
else if (count(listitems)) > 5)
  complicatedVariable = "moderately long list";
else
  complicatedVariable = "fairly short list";
```

Notice that a variable can be any of the XPath or XSLT variable types, including a *[1.0]* node-set or *[2.0]* sequence. When we call the `processListItems` template, the parameter we pass to it is a variable containing the node-set of all the `<listitem>` elements in our document. Inside the `processListItems` template, our variable (which is now technically a parameter) can be used inside an `<xsl:for-each>` element.

[2.0] For a final example, we'll use the `as` attribute to show how datatyping works. Here is a simple stylesheet:

```
<?xml version="1.0"?>
<!-- variable2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:variable name="numberOne" as="xs:integer" select="23"/>
  <xsl:variable name="numberTwo" as="xs:double" select="2.718281828459"/>
  <xsl:variable name="numberThree" as="xs:float" select="$numberTwo"/>
  <xsl:variable name="numberFour" as="xs:integer"
    select="xs:integer($numberTwo)"/>
  <xsl:variable name="dateValue" as="xs:date" select="xs:date('1995-04-21')"/>
  <xsl:variable name="whatever" as="xs:integer" select="xs:integer(42)"/>
```

```
  <xsl:template match="/">
    <xsl:call-template name="bob">
      <xsl:with-param name="whatever" select="xs:integer(8)"/>
    </xsl:call-template>
    <xsl:text>&#xA;whatever&#x9;</xsl:text>
    <xsl:value-of select="$whatever"/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

  <xsl:template name="bob">
    <xsl:param name="whatever" as="xs:integer"/>
    <xsl:variable name="whatever" as="xs:integer" select="xs:integer(4)"/>
    <xsl:variable name="whatever" as="xs:integer" select="xs:integer(98)"/>
    <xsl:text>Hello!  The values of your variables are:&#xA;</xsl:text>
    <xsl:text>&#xA;numberOne&#x9;</xsl:text>
    <xsl:value-of select="$numberOne"/>
    <xsl:text>&#xA;numberTwo&#x9;</xsl:text>
    <xsl:value-of select="$numberTwo"/>
    <xsl:text>&#xA;numberThree&#x9;</xsl:text>
    <xsl:value-of select="$numberThree"/>
    <xsl:text>&#xA;numberFour&#x9;</xsl:text>
    <xsl:value-of select="$numberFour"/>
    <xsl:text>&#xA;dateValue&#x9;</xsl:text>
    <xsl:value-of select="$dateValue"/>
    <xsl:text>&#xA;whatever&#x9;</xsl:text>
    <xsl:value-of select="$whatever"/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

When we run this stylesheet, here are the results:

```
Hello!  The values of your variables are:

numberOne       23
numberTwo       2.718281828459
numberThree     2.7182817
numberFour      2
dateValue       1995-04-21
whatever        98

whatever        42
```

Notice that we had to use the `xs:date` constructor to create a valid date, and that we lost several digits of accuracy when we copied the `double` value of `numberTwo` to the `float` value of `numberThree`. We also had to use the `xs:integer` constructor to cast the `double` value of `numberTwo` to the integer required by `numberFour`.

The `<xsl:variable>` elements here are all initialized with literal values, so if you change them to something that's not valid (assigning `'blue'` to the `xs:integer` value `$numberOne`, for example), the stylesheet won't run at all. If the values of the variables are set while the stylesheet is running, you'll get a fatal error if your stylesheet tries to set a variable with the wrong kind of data.

The variable $whatever is used several times in this stylesheet. Because variables and parameters can shadow each other in XSLT 2.0, the value of $whatever is the last value defined in the current context. Within the named template, the value of $whatever is 98, the last value assigned to that name. The fact that $whatever was previously assigned the values 4 (as a local variable), 8 (as a parameter), and 42 (as a global variable) doesn't matter. When the stylesheet returns to the calling template, the value of $whatever is now 42. The parameter passed to our named template and the local variables inside the named template are no longer in scope.

A complication of variables in XSLT 2.0: if you create a variable containing nodes and specify the datatype with the as attribute, the variable works as you would expect. However, if you don't use the as attribute, the variable is a new document node whose children are the nodes in the variable. Section 9.4 of the XSLT 2.0 spec discusses this in more detail. In general, it's best to use the as attribute.

<xsl:when>

Defines one branch of an <xsl:choose> element; <xsl:choose> is equivalent to an if-then-else statement.

Category

Subinstruction (<xsl:when> always appears as a child of an <xsl:choose> element).

Required Attribute

test

Contains a boolean expression that is evaluated. If the expression evaluates to true, the contents of the <xsl:when> element are processed; otherwise, the contents of <xsl:when> are ignored. When evaluating an <xsl:choose> element, the XSLT processor examines each <xsl:when> until it finds one whose test attribute evaluates to true. That <xsl:when> element is processed and all subsequent <xsl:when> elements are ignored. Within an <xsl:choose> element, at most one <xsl:when> element will be processed.

Optional Attributes

None.

Content

An XSLT template.

Appears in

The <xsl:choose> element only.

Defined in

[1.0] XSLT section 9.2, "Conditional Processing with `xsl:choose`."

[2.0] XSLT section 8.2, "Conditional Processing with `xsl:choose`."

Example

Here's an example that uses `<xsl:choose>` to select the background color for the rows of an HTML table. We cycle among four different values, using `<xsl:when>` to determine the value of the `style` attribute in the generated HTML document. Here's the XML document we'll use:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en_GB">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

And here's our stylesheet:

```
<?xml version="1.0"?>
<!-- when.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>
          <xsl:value-of select="list/title"/>
        </title>
      </head>
      <body style="font-family: sans-serif; color: white;">
        <h1 style="color: black;">
          <xsl:value-of select="list/title"/>
        </h1>
        <table border="1" cellpadding="5"
          style="font-weight: bold;">
          <xsl:for-each select="list/listitem">
            <tr>
              <td>
                <xsl:attribute name="style">
                  <xsl:choose>
                    <xsl:when test="position() mod 4 = 0">
                      <xsl:text>background: yellow; color: black;</xsl:text>
                    </xsl:when>
                    <xsl:when test="position() mod 4 = 1">
                      <xsl:text>background: blue;</xsl:text>
```

```
                    </xsl:when>
                    <xsl:when test="position() mod 4 = 2">
                      <xsl:text>background: white; color: black;</xsl:text>
                    </xsl:when>
                    <xsl:otherwise>
                      <xsl:text>background: black;</xsl:text>
                    </xsl:otherwise>
                  </xsl:choose>
                </xsl:attribute>
                <xsl:value-of select="."/>
              </td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

Here's the generated HTML document, which cycles through the various background colors:

```
<html>
  <head>
    <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Albums I've bought recently:</title>
  </head>
  <body style="font-family: sans-serif; color: white;">
    <h1 style="color: black;">Albums I've bought recently:</h1>
    <table border="1" cellpadding="5" style="font-weight: bold;">
      <tr>
        <td style="background: blue;">The Sacred Art of Dub</td>
      </tr>
      <tr>
        <td style="background: white; color: black;">Only the Poor Man Feel It</td>
      </tr>
      <tr>
        <td style="background: black;">Excitable Boy</td>
      </tr>
      <tr>
        <td style="background: yellow; color: black;">Aki Special</td>
      </tr>
      ...
    </table>
  </body>
</html>
```

The table cells with styles of `"background: yellow; color: black;"`, `"background: blue;"`, and `"background: white; color: black;"` are generated by `<xsl:when>` elements, while `"background: black;"` is generated by the `<xsl:otherwise>` element.

When rendered, our HTML document looks like Figure A-17.

Figure A-17. Generating different background colors with <xsl:when>

<xsl:with-param>

Defines a parameter to be passed to a template. When the template is invoked, values can be passed in for the parameter.

Category

Subinstruction; `<xsl:with-param>` always appears inside the `<xsl:apply-templates>` or `<xsl:call-template>` element.

[2.0] In XSLT 2.0, it can also appear inside the new `<xsl:apply-imports>` and `<xsl:next-match>` elements.

Description

`<xsl:with-param>` defines a parameter to be passed to a template. When a template is invoked, values can be passed in for its parameters. The value of a parameter can be defined in one of three ways:

- If the `<xsl:with-param>` element is empty and does not contain a `select` attribute, then no value is passed to the template.
- If the `<xsl:with-param>` element is empty and has a `select` attribute, the value of the parameter is the value of the `select` attribute.
- If the `<xsl:with-param>` element is not empty, the value of the parameter is the result of processing its contents.

If no value is passed to the template (`<xsl:with-param name="x"/>`), then the default value of the parameter, if any, is used instead. The default value of the parameter is

defined on the <xsl:param> element inside the <xsl:template> itself; see the description of the <xsl:param> element for more details.

Required Attribute

`name`
: Names this parameter.

Optional Attributes

`select`
: An XPath expression that defines the value of this parameter. It is a fatal error if <xsl:with-param> has a `select` attribute and contains content.

[2.0 – Schema] `as`
: Defines the datatype of this parameter. The datatype can be any of the built-in datatypes, or, if you have a schema-aware XSLT 2.0 processor, it can be a datatype defined in a schema.

[2.0] `tunnel`
: Defines whether this is a tunnel parameter. Valid values are `yes` and `no`, with `no` being the default.

Content

The <xsl:with-param> element can be empty or it can contain an XSLT template. It is a fatal error if <xsl:with-param> contains content and has a `select` attribute.

Appears in

<xsl:apply-templates>, <xsl:call-template>, *[2.0]* <xsl:apply-imports>, and *[2.0]* <xsl:next-match>.

Defined in

[1.0] XSLT section 11.6, "Passing Parameters to Templates."

[2.0] XSLT section 10.1.1, "Passing Parameters to Templates."

Example

Here is a stylesheet with a number of parameters. Notice that some parameters are global and defined outside the stylesheet:

```
<?xml version="1.0"?>
<!-- with-param.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:param name="favoriteNumber" select="23"/>
  <xsl:param name="favoriteColor"/>
```

```
  <xsl:template match="/">
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="list/title"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:variable name="listitems" select="list/listitem"/>
    <xsl:call-template name="processListItems">
      <xsl:with-param name="items" select="$listitems"/>
      <xsl:with-param name="color" select="'yellow'"/>
      <xsl:with-param name="number" select="$favoriteNumber"/>
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="processListItems">
    <xsl:param name="items"/>
    <xsl:param name="color" select="'blue'"/>

    <xsl:for-each select="$items">
      <xsl:value-of select="position()"/>
      <xsl:text>.  </xsl:text>
      <xsl:value-of select="."/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>

    <xsl:text>&#xA;</xsl:text>

    <xsl:text>Your favorite color is </xsl:text>
    <xsl:value-of select="$favoriteColor"/>
    <xsl:text>.</xsl:text>
    <xsl:text>&#xA;</xsl:text>
    <xsl:text>The color passed to this template is </xsl:text>
    <xsl:value-of select="$color"/>
    <xsl:text>.</xsl:text>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

We'll use this stylesheet to transform this document:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

Our stylesheet contains two global parameters (`favoriteNumber` and `favoriteColor`), and it defines a default value for `favoriteNumber`. The stylesheet also passes a parameter from the `match="/"` template to the `name="processListItems"` template; that parameter contains a node-set. Here are the results of the transformation:

```
Albums I've bought recently:
1.  The Sacred Art of Dub
2.  Only the Poor Man Feel It
3.  Excitable Boy
4.  Aki Special
5.  Combat Rock
6.  Talking Timbuktu
7.  The Birth of the Cool

Your favorite color is orange.
The color passed to this template is yellow.
```

Notice that when we call the template `processListItems`, we pass in three parameters, only two of which are defined inside the template. In XSLT 1.0, the undefined parameters are simply ignored; in XSLT 2.0, this is a fatal error. If we change the stylesheet to `<xsl:stylesheet version="2.0" ...>`, the stylesheet won't run at all. Also notice that the parameter `$items` is a node-set, so we can use the parameter in an `<xsl:for-each>` element.

To generate these results with Saxon, we use this command:

```
java net.sf.saxon.Transform albums.xml with-param.xsl favoriteColor=orange
```

To generate these results with Xalan, we use this command:

```
java org.apache.xalan.xslt.Process -in albums.xml -xsl with-param.xsl
  -param favoriteColor orange
```

(The command should be entered on a single line.) See "Global Parameters" in Chapter 5 for a complete discussion of global parameters and how you define them for various XSLT processors.

[2.0 – Schema] We'll wrap up with an example of `<xsl:with-param>` that uses schema support. We'll use a parameter whose datatype must be `zipcode`. Here's the XML Schema that defines the datatype:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- zip.xsd -->
<xsd:schema
  xmlns="http://www.oreilly.com/xslt/zip"
  targetNamespace="http://www.oreilly.com/xslt/zip"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:simpleType name="zipcode">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9]{5}(-[0-9]{4})?"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Here's the stylesheet that looks for a valid `zipcode`:

```
<?xml version="1.0"?>
<!-- with-param2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:zip="http://www.oreilly.com/xslt/zip"
  xmlns="http://www.oreilly.com/xslt/zip"
```

```
    xmlns:po="http://www.oreilly.com/xslt">

    <xsl:import-schema namespace="http://www.oreilly.com/xslt/zip"
      schema-location="zip.xsd" />

    <xsl:import-schema namespace="http://www.oreilly.com/xslt"
      schema-location="po.xsd" />

    <xsl:output method="text"/>

    <xsl:template match="/">
      <xsl:choose>
        <xsl:when
          test="/po:purchase-order/po:customer/po:address/po:zip
                castable as zip:zipcode">
          <xsl:call-template name="postalCode">
            <xsl:with-param name="zip" as="zip:zipcode"
              select="/po:purchase-order/po:customer/po:address/po:zip"/>
          </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
          <xsl:text>The &lt;zip&gt; element isn't valid!&#xA;</xsl:text>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:template>

    <xsl:template name="postalCode">
      <xsl:param name="zip" as="zip:zipcode"/>
      <xsl:text>The value </xsl:text>
      <xsl:value-of select="$zip"/>
      <xsl:text> is a valid Zip code!&#xA;</xsl:text>
    </xsl:template>

  </xsl:stylesheet>
```

When we use this template against a valid `<po:purchase-order>` whose `<zip>` element can be cast as a `zip:zipcode` value, we use `<xsl:with-param>` to pass a parameter of type `zip:zipcode`. We'll use this purchase order again:

```
  <?xml version="1.0"?>
  <!-- good-po.xml -->
  <purchase-order id="38292"   xmlns="http://www.oreilly.com/xslt">
    <date year="2001" month="6" day="19"/>
    <customer id="4738" level="Platinum">
      <address type="business">
        ...
        <zip>48392</zip>
      </address>
    </customer>
    <items>
    ...
    </items>
  </purchase-order>
```

The less-than-exciting results look like this:

```
The value 48392 is a valid Zip code!
```

Notice that we imported two schemas, one for the `zip:zipcode` datatype and one for the `<po:purchase-order>` element, although these could have been in the same file. If the elements in the purchase order are in a namespace, we would have to use the `po:` prefix on all the elements in the purchase order or else use `xpath-default-namespace`. Our stylesheet uses the XPath `castable as` operator to verify that the value of the `<po:zip>` element can be cast as a `zip:zipcode` value. That means we use the `<xsl:with-param>` and `<xsl:param>` elements only if the data is valid. If it is, we create the parameter, knowing that the `as` attributes will be satisfied.

One final point: the `as` attribute can only contain sequence types. Specifically, the `as` attribute can't refer to complex types declared in an XML Schema. We could rewrite our purchase order schema to have a `purchase-order` complex type. To use it on the `as` attribute, we would have to code `as="element(*, po:purchase-order)"`.

APPENDIX B
XPath Reference

This appendix contains reference information from the XPath specification, including node types, axes, operators, and datatypes.

XPath Node Types

There are seven types of nodes in XPath. (They're called *node kinds* in XPath 2.0.) We'll stick to the reference material here; for more information on the different node types, see our earlier discussion of "The XPath Data Model" in Chapter 3.

The Root Node

The root node is the root of the tree. Unlike all other nodes, it does not have a parent. Its children are the root element for the document, as are any comments or processing instructions that appear outside the document element. The root node does not have an expanded name. It is known as the *document node* in XPath 2.0.

Element Nodes

Each element in the original XML document is represented by an element node. The expanded name of the element is its local name, combined with any namespace that is in effect for the element. You can access the different parts of the element name with the `name( )`, `local-name( )`, and `namespace-uri( )` functions. Here is an element from an XML document:

```
    <xyz:report xmlns:xyz="http://www.xyz.com/">
```

The values of the three functions for this element node are:

```
name( )
    xyz:report
local-name( )
    report
```

```
namespace-uri( )
    http://www.xyz.com/
```

Attribute Nodes

Attributes of elements in the XML document become XPath attribute nodes. An attribute has an expanded name, just as an element node has. The attribute nodes of a given element node are the attributes explicitly coded on the XML element and any attributes defined with default values in the DTD.

Taking a different approach from the Document Object Model, an element node is the parent of its attributes, although the attributes are not the children of the element. In other words, selecting all the children of an element node does not select any attribute nodes that the element node might have.

Text Nodes

Text nodes simply contain text from an element. If the original text in the XML document contained character or entity references, they are resolved before the XPath text node is created. Similarly, any existing CDATA sections appear as text nodes. You have no way of knowing if a given portion of a text node was originally a character or entity reference or a CDATA section.

Comment Nodes

A comment node is also very simple; it contains some text. Every comment in the source document (except for any comments in the DTD) becomes a comment node. The text of the comment node (returned with `<xsl:value-of select=".">`) contains everything inside the comment except the opening `<!--` and the closing `-->`.

Processing-Instruction Nodes

A processing-instruction node has two parts: a name (returned by the `name( )` function) and a string value. The string value is everything after the name, including the whitespace, but not including the `?>` that closes the processing instruction.

Namespace Nodes

Namespace nodes are almost never used in XSLT stylesheets; they exist primarily for the XSLT processor's benefit. One thing to keep in mind is that the declaration of a namespace (such as `xmlns:auth="http://www.authors.net"`), even though it looks like an attribute in the XML source, becomes a namespace node and not an attribute node. Namespace nodes exist for both the namespace prefixes that are defined and any default namespaces.

XPath Node Tests

XPath defines several node tests that can be used to select nodes from the source tree. These node tests allow you to select nodes that can't be selected any other way. (Although they look and work like functions, they are technically node tests.) The node tests are described here:

`text( )`
: Selects all the text-node children of the context node.

`comment( )`
: Selects all the comment-node children of the context node.

`processing-instruction( )`
: Selects all the processing-instruction children of the context node. Unlike the other node tests defined here, `processing-instruction( )` can have an optional argument; `processing-instruction('xml-stylesheet')` selects all processing instructions with a name of `xml-stylesheet`.

`node( )`
: Returns all nodes, regardless of type. Using this node test selects all element nodes, attribute nodes, processing-instruction nodes, etc. (Be aware that using `node( )` on the child axis does *not* return any attribute nodes, because attributes are not considered child nodes.)

[2.0] `attribute( )`
: Returns any attribute. When used with an attribute name (`attribute(public-domain)`), it returns all attributes with the specified name. When used with an attribute name and a datatype (`attribute(public-domain, xs:boolean)`), it returns all attributes with that specified name and datatype. Finally, when used with a wildcard for the attribute name (`attribute(*, xs:boolean)`), it returns all attributes with the specified datatype.

[2.0] `element( )`
: Returns any element. When used with an element name (`element(author)`), it returns all elements with the specified name. When used with an element name and a datatype (`element(year-of-birth, xs:gYear)`), it returns all elements with that specified name and datatype. Finally, when used with a wildcard for the element name (`element(*, xs:gYear)`), it returns all elements with the specified datatype.

[2.0] `schema-element(name)`
: Given the name of an element globally declared in an XML Schema, returns the elements with the same name and the same datatype as the schema-defined element. It also returns all the elements in the specified element's substitution group. Appendix D discusses "Substitution Groups" in more detail.

[2.0] `document-node( )`
: Matches document nodes. The node test can include a name; the test `document-node(element(sonnet))` returns a document node whose root element is `<sonnet>`.

[2.0] Although `item( )` looks like a node test, it is only used as a datatype. For example, the variable `<xsl:variable name="something" as="item( )">` defines a variable that can contain a single value. The variable can contain any node or atomic type.

XPath Axes

The XPath specification defines 13 different axes; each axis contains various nodes. The nodes that are in a given axis depend on the context node. All 13 axes, excerpted from our more involved discussion in "The XPath Data Model" in Chapter 3, are listed here.

`child` *axis*
: Contains the children of the context node. As we've already mentioned, the XPath expressions `child::lines/child::line` and `lines/line` are equivalent. If an XPath expression (such as `sonnet`) doesn't have an axis specifier, the `child` axis is used by default.

`parent` *axis*
: Contains the parent of the context node, if there is one. (If the context node is the root node, the `parent` axis returns an empty node-set.) As a step in an XPath expression, the `parent` axis can be abbreviated with the double period (`..`); this moves up to the current node's parent. If the `<first-name>` and `<last-name>` elements are both children of the `<author>` element, and the context node is the `<first-name>` element, the expressions `../last-name`, `parent::author/last-name` and `parent::*/last-name` are equivalent. If the context node does not have a parent, this axis returns an empty node-set.

`self` *axis*
: Contains the context node itself. As a step in an XPath expression, the `self` axis can be abbreviated with a single period (`.`). The expressions `.`, `self::node( )`, and `self::*` are equivalent in XSLT 1.0.

  [2.0] In XSLT 2.0, the `self` axis selects the context *item*, which might not be a node. If the context item is an atomic value, the expressions `self::node( )` and `self::*` cause the XSLT processor to raise an error. In this case, the only way to access the `self` axis is with a period. If the context item is a node, the `self` axis works just as it did in XSLT 1.0.

`attribute` *axis*
: Contains the attributes of the context node. If the context node is not an element node, this axis is empty. The `attribute` axis can be abbreviated with the at sign (`@`). The expressions `attribute::type` and `@type` are equivalent.

`ancestor` *axis*
: Contains the parent of the context node, the parent's parent, and so on. The `ancestor` axis always contains the root node, unless the context node is the root node.

`ancestor-or-self` *axis*
: Contains the context node, its parent, its parent's parent, and so on. This axis always includes the root node.

`descendant` *axis*
: Contains all children of the context node, all children of all the children of the context node, and so on. Be aware that the `descendant` axis does not include any attribute or namespace nodes. (As we discussed earlier, an attribute node has an element node as its parent, even though the attribute node is not considered a child of its parent.)

`descendant-or-self` *axis*
: Contains the context node and all children of the context node, all children of all the children of the context node, and so on.

`preceding-sibling` *axis*
: Contains all of the preceding siblings of the context node—in other words, all nodes that have the same parent as the context node and appear before the context node in the XML document. If the context node is an attribute node or a namespace node, the `preceding-sibling` axis is empty.

`following-sibling` *axis*
: Contains all of the following siblings of the context node—in other words, all nodes that have the same parent as the context node and appear after the context node in the XML document. If the context node is an attribute node or a namespace node, the `following-sibling` axis is empty.

`preceding` *axis*
: Contains all nodes that appear before the context node in the document, except any ancestors, attribute nodes, and namespace nodes.

`following` *axis*
: Contains all nodes that appear after the context node in the document, except any descendants, attribute nodes, and namespace nodes.

`namespace` *axis*
: Contains the namespace nodes of the context node. If the context node is not an element node, this axis is empty.

  [2.0] The namespace axis is deprecated in XPath 2.0.

The five axes `ancestor`, `descendant`, `following`, `preceding`, and `self` partition everything in the XML document (with the exception of any attribute or namespace nodes). Any node in the XPath tree appears in one of these five axes, and the five axes do not overlap.

The XPath Context

The context in an XPath expression consists of several things:

Context node
: The node currently being evaluated. *[2.0]* If the context item is a node, the context node is the same as the context item. If the context item is an atomic value, the context node is undefined.

[2.0] Context item
: The item currently being evaluated. This is the equivalent of the context node in XPath 1.0; the name reflects the fact that the context can be focused on an atomic value instead of a node.

Context position
: A nonzero positive integer that indicates the position of the context node within the set of context nodes. The XPath function `position()` returns the context position.

Context size
: A nonzero positive integer that indicates the number of nodes in the current context. The XPath function `last()` returns the context size.

Variable bindings
: A set of variables that are in scope for the current context. Each one is represented by a variable name and an object that represents its value.

Functions
: A set of functions visible to the current context. Each function is represented by a mapping between a function name and the actual code to be invoked. Each function takes zero or more arguments and returns a single result. XPath defines a number of core functions that are always available; XSLT defines additional functions that go beyond those defined in the XPath specification. Any extension functions defined in the stylesheet are visible as well.

Namespace declarations
: The set of namespace declarations visible to the current context. Each one consists of a namespace prefix and the URI with which it is associated.

[2.0] Default namespace
: The default namespace is defined by the `xpath-default-namespace` attribute. If no such attribute exists on any elements that enclose the context item, the default namespace is null, regardless of any default namespace declarations (`xmlns="..."`) in the stylesheet.

[2.0] Documents and collections
: The XPath 2.0 context also includes information about available documents, available collections, and the default collection. The functions `doc()`, `doc-available()`, and `collection()` functions work with documents and collec-

tions of nodes. See the discussions of the [2.0] `doc( )` function, the [2.0] `doc-available( )` function, and the [2.0] `collection( )` function for more details.

[2.0] Miscellaneous information
Given the added complexity of XPath 2.0 and XSLT 2.0, there are a number of other things stored in the context. Many of them apply only in certain situations (the current group is meaningful only when we're grouping data, for example), so they're listed briefly here:

- The current template rule.
- The current template mode.
- The current group and current grouping key.
- The current captured substrings (used within `<xsl:analyze-string>`).
- The output state, which indicates whether output is being written to a result tree or a data structure. For example, within an `<xsl:variable>` element, the output state is to a data structure.
- The implicit timezone.
- The set of named keys, used by the `key( )` function.
- The set of named decimal formats, used by the `format-number( )` function.
- The values of all the system properties, used by the `system-property( )` function.
- The set of available elements, used by the `element-available( )` function.
- The set of all known collations.
- The default collation.
- The base URI of the containing element. This is returned by the XPath `base-uri( )` function.
- The set of in-scope schema definitions.

XPath 1.0 Datatypes

XPath 1.0 and XSLT 1.0 define five datatypes, described in the list that follows. The `result tree fragment` type is defined by XSLT 1.0 and is specific to transformations; the other four are defined by XPath and are generic to any technology that uses XPath. The four XPath datatypes are tersely defined in section 1 of the XPath specification; section 11.1 of the XSLT specification defines result tree fragments.

`node-set`
A set of nodes. The set can be empty or it can contain any number of nodes.

[2.0] In XSLT 2.0, the `node-set` has been replaced by the sequence.

`boolean`
: The value `true` or `false`. Be aware that the strings `true` and `false` have no special meaning or value in XPath. If you need to use the boolean values themselves, use the functions `true( )` and `false( )`.

`number`
: A floating-point number. All numbers in XPath and XSLT 1.0 are implemented as floating-point numbers; the `integer` or `int` datatype does not exist in XPath and XSLT 1.0. To be specific, all numbers are implemented as IEEE 754 floating-point numbers, the same standard used by the Java `float` and `double` primitive types. In addition to ordinary numbers, there are five special values for numbers: positive and negative infinity, positive and negative zero, and `NaN`, the special symbol for anything that is not a number.

`string`
: Zero or more characters, as defined in the XML specification.

`result tree fragment`
: A temporary tree. You can create one with an `<xsl:variable>` element that uses content (instead of the `select` attribute) to initialize its value. A result tree fragment can be copied to the result tree with the `<xsl:copy-of>` element. It may also be converted to a string with the `<xsl:value-of>` element.

  [2.0] In XSLT 2.0, the `result tree fragment` datatype no longer exists. There is no difference between a tree constructed from an input document and a tree constructed using a variable.

[2.0] XPath 2.0 Datatypes

XML Schema defines 19 primitive datatypes, and five others (`xs:anyAtomicType`, `xs:untyped`, `xs:untypedAtomic`, `xs:dayTimeDuration`, and `xs:yearMonthDuration`) were added to the XML Schema namespace by the XQuery 1.0 and XPath 2.0 Data Model spec. We'll review those briefly here.

XPath 1.0 used the `node-set` datatype as its basic data structure; XPath 2.0 uses sequences. Like a `node-set`, a *sequence* can contain the node types we covered earlier in this appendix, but it can also contain atomic values of the types listed here.

Here are the 24 datatypes:

`xs:anyAtomicType`
: The base type for all primitive atomic types, such as `xs:integer` or `xs:string`. This datatype was added by XQuery 1.0 and XPath 2.0.

`xs:anyURI`
: A Uniform Resource Identifier. The value can be absolute or relative, and it can also have a fragment reference identifier. The value should follow the rules for URI syntax as defined in RFC 2396 and amended in RFC 2732.

`xs:base64Binary`
: Arbitrary binary data represented using the Base64 alphabet defined in RFC 2045. Legal characters are the basic alpha characters and digits `[a-zA-Z0-9]`, along with the plus sign (`+`), the forward slash (`/`), and the equals sign (`=`). An `xs:base64Binary` value can also contain any number of whitespace characters.

`xs:boolean`
: The value `true` or `false`. In XSLT and XPath, the strings `"true"` and `"false"` have no special meaning. To generate boolean values, use the functions `true( )` and `false( )` instead.

`xs:date`
: A date with four components: year, month, day, and an optional timezone. The format of a complete `xs:date` value is `1995-04-21-05:00`, where `1995` is the year, `04` is the month, `21` is the day, and `-05:00` is the timezone. An optional minus sign can appear before the year, indicating a date before the current era.

  Be aware that the values for month and day must be two digits long, and the value for the year must be at least four digits long. The value `1995-4-21` is illegal. The seconds can be specified to any number of decimal places.

`xs:dateTime`
: A date and time with seven components: year, month, day, hours, minutes, seconds, and an optional timezone. The format of a complete `xs:dateTime` value is `1995-04-21T00:05:32.6-05:00`, where `1995` is the year, `04` is the month, `21` is the day, `00` is the hours, `05` is the minutes, `32.6` is the seconds, and `-05:00` is the timezone. An optional minus sign can appear before the year, indicating a date and time before the current era.

  Be aware that the values for month, day, hours, minutes and seconds must be two digits long, and the value for the year must be at least four digits long. The value `1995-4-21T00:5:32.6` is illegal. The seconds can be specified to any number of decimal places.

`xs:dayTimeDuration`
: A new datatype defined by XQuery 1.0 and XPath 2.0. It is derived from `xs:duration` and can only contain the days, hours, minutes, and seconds components of an `xs:duration`. For example, `P2DT4H32M12.83S` is a duration of 2 days, 4 hours, 32 minutes, and 12.83 seconds.

`xs:decimal`
: A sequence of digits with a decimal point. An `xs:decimal` is allowed to have a leading sign (`-` or `+`). In addition, if there is no fractional portion of the number, the decimal point can be omitted. Sample `xs:decimal` values are `210`, `12678967.543233`, `-1.23`, and `+100000.00`.

`xs:double`
: A number based on the IEEE double-precision 64-bit floating-point type. Like `xs:decimal`, `xs:double` values can have a leading sign (`-` or `+`). If there is no fractional

portion of the number, the decimal point can be omitted. An `xs:double` can also have an exponent, represented by `E` or `e`, followed by the exponent. The exponent must be an integer.

There are three special values for `xs:double`: `INF`, `-INF`, and `NaN`. They represent positive infinity, negative infinity, and not a number. Sample `xs:double` values are `-1E4`, `1267.43233E12`, `12.78e-2`, `12`, `-0`, `0`, and `INF`.

`xs:duration`
: An `xs:duration` is a six-dimensional period of time, with the six dimensions of years, months, days, hours, minutes and seconds. The string `P2Y3M2DT7H52M23.8S` is a duration of 2 years, 3 months, 2 days, 7 hours, 52 minutes, and 23.8 seconds. The `P` character is always required. The `T` character separates the date portion from the time portion and can be omitted if none of the time components are specified. In other words, `P2Y3M` is a duration of 2 years and 3 months. If any component of the duration is zero, it may be omitted, although at least one portion of the duration must be specified. For example, `PT2H` is a duration of 2 hours, as is `P0Y0M0DT2H0M0S`.

`xs:float`
: A number based on the IEEE single-precision 32-bit floating-point type. Like `xs:decimal`, `xs:float` values can have a leading sign (`-` or `+`). If there is no fractional portion of the number, the decimal point can be omitted. `xs:float` can also have an exponent, represented by `E` or `e`, followed by the exponent. The exponent must be an integer.

  There are three special values for `xs:float`: `INF`, `-INF`, and `NaN`. They represent positive infinity, negative infinity, and not a number. Sample `xs:float` values are `-1E4`, `1267.43233E12`, `12.78e-2`, `12`, `-0`, `0`, and `INF`.

`xs:gDay`
: A specific day. It is specified as three dashes followed by the two-digit day, such as `---21`. A timezone is allowed.

`xs:gMonth`
: A specific month. It is specified as two dashes followed by the two-digit month, such as `--04`. A timezone is allowed.

`xs:gMonthDay`
: A specific day in a specific month. An `xs:gMonthDay` is specified as two dashes, the two-digit month, a dash, and the two-digit day, such as `--04-21`. A timezone is allowed.

`xs:gYear`
: A specific year. It is specified as number with at least four digits, such as `1995`. The year can have a leading minus sign, and a timezone is allowed.

`xs:gYearMonth`
: A specific month in a specific year. It is specified in the format `1995-04`. The year value must be at least four digits and can have a leading minus sign. The month value must be two digits. A timezone is allowed.

`xs:hexBinary`
: Arbitrary hex-encoded binary data. Each binary octet in the original data is represented as two hexadecimal digits (`[a-fA-F0-9]`).

`xs:NOTATION`
: This rarely used datatype represents the `NOTATION` attribute type defined in the XML specification.

`xs:QName`
: Consists of a namespace name (sometimes defined with a namespace prefix) and a local name.

`xs:string`
: A sequence of characters.

`xs:time`
: A time with four components: hours, minutes, seconds, and an optional timezone. The format of a complete `xs:time` value is `00:05:32.6-05:00`, where `00` is the hours, `05` is the minutes, `32.6` is the seconds, and `-05:00` is the timezone.

  Be aware that the values for hours, minutes, and seconds must be two digits long. The seconds can be specified to any number of decimal places.

`xs:untyped`
: The datatype of an element that has not been validated. If an element is validated in skip mode, that element's type is `xs:untyped` as well. This datatype was added by XQuery 1.0 and XPath 2.0.

`xs:untypedAtomic`
: An untyped atomic value. Text that has not been assigned a more specific type is considered `xs:untypedAtomic`, as is any attribute validated in skip mode. Text from an input document that did not have a schema is `xs:untypedAtomic`, for example. This datatype was added by XQuery 1.0 and XPath 2.0.

`xs:yearMonthDuration`
: A new datatype defined by XQuery 1.0 and XPath 2.0. It is derived from `xs:duration` and can contain only the year and month components of an `xs:duration`. For example, `P2Y3M` is a duration of two years and three months.

Be aware that you do not have to have a schema-aware XSLT 2.0 processor to use these basic datatypes. You can use the `cast as`, `castable as`, and `instance of` operators with all of these datatypes. What you can't do without a schema-aware XSLT processor is define your own datatypes in a schema, and then use those datatypes with these operators.

Operators and Keywords

Here is the complete list of operators and keywords in XPath:

`!=` *(not equal)*
: Compares its two operands and returns `true` if the first operand is not equal to the second. (Complete details on how comparisons work are in the section "Boolean Operators" in Chapter 3.)

`( )`
: Contains a parenthesized expression. In addition to using parentheses in mathematical expressions, parentheses are required around test conditions in the `if` operator.

`*` *(occurrence indicator)*
: Represents zero or more of an item.

`*` *(multiplication)*
: Multiplies its two operands together. In XPath 1.0, the two operands must be numbers or values that can be converted to numbers.

  [2.0] In XPath 2.0, we can multiply `xs:yearMonthDuration`s and `xs:dayTimeDurations` by numeric values. Valid combinations are:

  - `xs:yearMonthDuration * xs:double`
  - `xs:dayTimeDuration * xs:double`

`+` *(occurrence indicator)*
: Represents zero or one of an item.

`+` *(addition)*
: Adds its two operands. In XPath 1.0, the two operands must be numbers or values that can be converted to numbers.

  [2.0] In XPath 2.0, we can add dates, times, and durations in addition to numeric values. Valid combinations are:

  - `xs:yearMonthDuration + xs:yearMonthDuration`
  - `xs:dayTimeDuration + xs:dayTimeDuration`
  - `xs:dateTime + xs:yearMonthDuration`
  - `xs:dateTime + xs:dayTimeDuration`
  - `xs:date + xs:yearMonthDuration`
  - `xs:date + xs:dayTimeDuration`
  - `xs:time + xs:dayTimeDuration`

`+` *(unary plus)*
: Returns its operand with the sign unchanged. This operator doesn't change its operand at all.

[2.0] `,` *(sequence operator)*
: The comma operator concatenates items into a sequence. For example, using `select="((1, 2, 3), (4, 5), 6)"` creates the new sequence `(1, 2, 3, 4, 5, 6)`.

`-` *(unary minus)*
: Returns the negation of its operand.

`-` *(subtraction)*
: Subtracts its second operand from the first. In XPath 1.0, the two operands must be numbers or values that can be converted to numbers.

  [2.0] In XPath 2.0, we can subtract dates, times, and durations in addition to numeric values. Valid combinations are:

  - `xs:yearMonthDuration - xs:yearMonthDuration`
  - `xs:dayTimeDuration - xs:dayTimeDuration`
  - `xs:dateTime - xs:yearMonthDuration`
  - `xs:dateTime - xs:dayTimeDuration`
  - `xs:date - xs:yearMonthDuration`
  - `xs:date - xs:dayTimeDuration`
  - `xs:time - xs:dayTimeDuration`

`/` *(location step)*
: Represents a step in an location path.

`//` *(location step)*
: Represents zero or more levels in a location path.

`<` *(less than)*
: Compares its two operands and returns `true` if the first operand is less than the second. (Complete details on how comparisons work are in the section "Boolean Operators" in Chapter 3.)

[2.0]`<<` *(node-before)*
: Compares two nodes and returns `true` if the first node appears before the second in the source document. Returns `false` otherwise, including the case in which the two nodes are the same. Because an attribute in XSLT can't contain a left bracket, this operator must be coded `<<`.

`<=` *(less than or equal to)*
: Compares its two operands and returns `true` if the first operand is less than or equal to the second. (Complete details on how comparisons work are in the section "Boolean Operators" in Chapter 3.)

`=` *(equal to)*
: Compares its two operands and returns `true` if the first operand is equal to the second. (Complete details on how comparisons work are in the section "Boolean Operators" in Chapter 3.)

`>` *(greater than)*
: Compares its two operands and returns `true` if the first operand is greater than the second. (Complete details on how comparisons work are in the section "Boolean Operators" in Chapter 3.)

[2.0] `>>` *(node-after)*
: Compares two nodes and returns `true` if the first node appears after the second in the source document. Returns `false` otherwise, including the case in which the two nodes are the same.

`>=` *(greater than or equal to)*
: Compares its two operands and returns `true` if the first operand is greater than or equal to the second. (Complete details on how comparisons work are in the section "Boolean Operators" in Chapter 3.)

`?` *(occurrence indicator)*
: Represents zero or one of an item.

`[ ]`
: Contains a predicate in an XPath expression.

`|` *(union)*
: Compares two node-sets and returns a node-set containing all of the nodes from both node-sets. *[2.0]* In XPath 2.0, the vertical bar is identical to the `union` operator.

`and`
: Given two expressions, returns `true` if both are `true`. Returns false if either is `false`.

[2.0] `cast as`
: Casts a value to another type. Be aware that `cast as` causes a fatal error if the value can't be cast to the new type. For example, `"'Lily' cast as xs:integer"` causes a fatal error. To check whether `cast as` will work, use the `castable as` operator first.

[2.0] `castable as`
: Determines whether a value can be cast as another value type. The expression `"'3' castable as xs:integer"` is `true`. As you'd expect, `"'Lily' castable as xs:integer"` is `false`. Unlike the `cast as` operator, `castable as` doesn't actually cast the value. It simply lets us know whether the cast will work.

`div` *(division)*
: Divides its first operand by the second. In XPath 1.0, the two operands must be numbers or values that can be converted to numbers.

  [2.0] In XPath 2.0, we can divide durations:

  - `xs:yearMonthDuration div xs:double`
  - `xs:yearMonthDuration div xs:yearMonthDuration`
  - `xs:dayTimeDuration div xs:double`
  - `xs:dayTimeDuration div xs:dayTimeDuration`

[2.0] `eq` *(equal to)*
Compares two atomic values and returns `true` if the two values are equal. The `eq` operator can be used to compare the following *datatypes*:

- Numeric values (`xs:decimal`, `xs:double`, `xs:float`, `xs:integer`)
- String values
- Boolean values
- Durations (`xs:duration`, `xs:yearMonthDuration`, `xs:dayTimeDuration`)
- Dates and times (`xs:date`, `xs:time`, `xs:dateTime`)
- Parts of dates (`xs:gYear`, `xs:gYearMonth`, `xs:gMonth`, `xs:gMonthDay`, `xs:gDay`)
- QNames (`xs:QName`)
- Binary data (`xs:hexBinary`, `xs:base64Binary`)
- Notations (`xs:NOTATION`)

[2.0] `every`
Given a sequence and a test condition, returns `true` if every item in the sequence satisfies the condition. The operator is written as `every $x in $sequence satisfies [test condition for $x]`.

[2.0] `except`
Compares two sequences of nodes and returns a sequence containing the nodes that appear in the first sequence but not the second. All duplicate nodes are removed. The `except` operator compares the nodes themselves, not their values.

[2.0] `for`
This operator is an iterator across a sequence. The operator, written as `for $x in $sequence return ...`, iterates through all the values in `$sequence`. For each iteration through the sequence, the value `$x` represents the current value.

[2.0] `ge` *(greater than or equal to)*
Compares two atomic values and returns `true` if the first value is greater than or equal to the second. The `ge` operator can be used to compare the following datatypes:

- Numeric values (`xs:decimal`, `xs:double`, `xs:float`, `xs:integer`)
- String values
- Boolean values
- Durations (`xs:yearMonthDuration` and `xs:dayTimeDuration`, but *not* `xs:duration`)
- Dates and times (`xs:date`, `xs:time`, `xs:dateTime`)

[2.0] `gt` *(greater than)*
Compares two atomic values and returns `true` if the first value is greater than the second. The `gt` operator can be used to compare the following datatypes:

- Numeric values (`xs:decimal`, `xs:double`, `xs:float`, `xs:integer`)
- String values

- Boolean values
- Durations (`xs:yearMonthDuration` and `xs:dayTimeDuration`, but *not* `xs:duration`)
- Dates and times (`xs:date`, `xs:time`, `xs:dateTime`)

[2.0] `idiv` *(integer division)*
: Divides its first operand by the second. Any remainder is discarded, and the integer portion is returned.

[2.0] `if`
: Evaluates an expression, then performs one action or another depending on whether the expression evaluated to `true` or `false`. The expression must be (in parentheses), and there must be a `then` and an `else`.

[2.0] `instance of`
: Determines whether an argument is an instance of a particular datatype. The expression `"3 instance of xs:integer"` is `true`; `"'j' instance of xs:integer"` is `false`.

[2.0] `intersect`
: Compares two sequences of nodes and returns a sequence containing the nodes that appear in both sequences. All duplicate nodes are removed. The `intersect` operator compares the nodes themselves, not their values.

[2.0] `is`
: Compares two nodes and returns `true` if they are *the same node*. If two different nodes have the same values, `node1 is node2` returns `false`. The `is` operator compares the nodes, not their values.

[2.0] `le` *(less than or equal to)*
: Compares two atomic values and returns `true` if the first value is less than or equal to the second. The `le` operator can be used to compare the following datatypes:
  - Numeric values (`xs:decimal`, `xs:double`, `xs:float`, `xs:integer`)
  - String values
  - Boolean values
  - Durations (`xs:yearMonthDuration` and `xs:dayTimeDuration`, but *not* `xs:duration`)
  - Dates and times (`xs:date`, `xs:time`, `xs:dateTime`)

[2.0] `lt` *(less than)*
: Compares two atomic values and returns `true` if the first value is less than the second. The `lt` operator can be used to compare the following datatypes:
  - Numeric values (`xs:decimal`, `xs:double`, `xs:float`, `xs:integer`)
  - String values
  - Boolean values
  - Durations (`xs:yearMonthDuration` and `xs:dayTimeDuration`, but *not* `xs:duration`)
  - Dates and times (`xs:date`, `xs:time`, `xs:dateTime`)

`mod` *(modulus)*
: Divides its first operand by the second and returns the remainder.

[2.0] `ne` *(not equal to)*
: Compares two atomic values and returns `true` if the two values are *not* equal. The `ne` operator can be used to compare the following datatypes:

  - Numeric values (`xs:decimal`, `xs:double`, `xs:float`, `xs:integer`)
  - Boolean values
  - String values
  - Durations (`xs:duration`, `xs:yearMonthDuration`, `xs:dayTimeDuration`)
  - Dates and times (`xs:date`, `xs:time`, `xs:dateTime`)
  - Parts of dates (`xs:gYear`, `xs:gYearMonth`, `xs:gMonth`, `xs:gMonthDay`, `xs:gDay`)
  - QNames (`xs:QName`)
  - Binary data (`xs:hexBinary`, `xs:base64Binary`)
  - Notations (`xs:NOTATION`)

`or`
: Given two expressions, returns `true` if either or both are `true`. Returns `false` if both are `false`.

[2.0] `some`
: Given a sequence and a test condition, returns `true` if at least one item in the sequence satisfies the condition. The operator is written `some $x in $sequence satisfies [test condition for $x]`.

[2.0] `to` *(range)*
: Creates a sequence of integers. The expression `1 to 10` creates a sequence of 10 integers, ordered `1, 2, 3...`.

[2.0] `treat as`
: Disables XPath 2.0's static type checking. If `$a` is of type `xs:integer*` and a function needs a parameter of type `xs:integer`, we can use the expression `$a treat as xs:integer` to get around the static type checking. At runtime, if `$a` is a sequence of one `xs:integer`, `treat as` works without any errors. On the other hand, if `$a` is anything else, including an empty sequence, a runtime error occurs.

[2.0] `union`
: Compares two sequences of nodes and returns a sequence containing all of the nodes from both sequences. All duplicate nodes are removed. The `union` operator compares the nodes themselves, not their values.

Operator Precedence—XPath 1.0

Here is the precedence of operators in XPath 1.0, arranged from lowest to highest:

- `or`
- `and`
- `=`, `!=`, `<`, `<=`, `>`, `>=`
- `+` (addition), `-` (subtraction)
- `*` (multiplication), `div`, `mod`
- `|`
- `-` (unary minus), `+` (unary plus)
- `/`, `//`
- `[ ]`, `( )`, `{ }`

[2.0] Operator Precedence—XQuery 1.0 and XPath 2.0

Here is the precedence of operators in XQuery 1.0 and XPath 2.0, arranged from lowest to highest:

- `,` (comma)
- `for`, `some`, `every`, `if`
- `or`
- `and`
- `eq`, `ne`, `lt`, `le`, `gt`, `ge`, `=`, `!=`, `<`, `<=`, `>`, `>=`, `is`, `<<`, `>>`
- `to`
- `+` (addition), `-` (subtraction)
- `*` (multiplication), `div`, `idiv`, `mod`
- `union`, `|`
- `intersect`, `except`
- `instance of`
- `treat as`
- `castable as`
- `cast as`
- `-` (unary minus), `+` (unary plus)
- `?`, `*` (occurrence indicator), `+` (occurrence indicator)
- `/`, `//`
- `[ ]`, `( )`, `{ }`

APPENDIX C

XSLT, XPath, and XQuery Function Reference

This section lists all functions defined by XSLT 1.0 and 2.0, XPath 1.0 and 2.0, and XQuery 1.0.

Kinds of Functions

Including the new functions added by XSLT 2.0 and XPath 2.0, there are now more than 100 useful functions. The following presents a categorized list of them.

Accessor Functions

XPath 2.0 provides accessor functions to expose certain properties of items, sequences, and documents:

- [2.0] `base-uri( )`
- [2.0] `data( )`
- [2.0] `document-uri( )`
- [2.0 – Schema] `nilled( )`
- [2.0] `node-name( )`
- `string( )`

Boolean Functions

These functions work with Boolean values:

- `false( )`
- `not( )`
- `true( )`

Constructor Functions

There are two functions for creating `xs:dateTime` and `xs:QName` values:

- [2.0] `dateTime( )`
- [2.0] `QName( )`

Context Functions

The XPath *context* has a number of properties. These functions allow us to access those properties:

- [2.0] `collection( )`
- `current( )`
- [2.0] `current-date( )`
- [2.0] `current-dateTime( )`
- [2.0] `current-time( )`
- [2.0] `default-collation( )`
- `element-available( )`
- `function-available( )`
- [2.0] `implicit-timezone( )`
- `last( )`
- `position( )`
- [2.0] `static-base-uri( )`
- [2.0] `type-available( )`

The functions `element-available( )`, `function-available( )`, and [2.0] `type-available( )` are actually defined in XSLT 2.0.

Cross-Referencing and Grouping Functions

These functions help you resolve unique identifiers and references between elements and attributes:

- [2.0] `current-group( )`
- [2.0] `current-grouping-key( )`
- `generate-id( )`
- `id( )`

- [2.0] `idref( )`
- `key( )`

Date, Time, and Duration Functions

These functions let you work with date, time and duration values. These are the XML Schema types `xs:date`, `xs:dateTime`, `xs:dayTimeDuration`, `xs:duration`, `xs:time`, and `xs:yearMonthDuration`:

- [2.0] `adjust-date-to-timezone( )`
- [2.0] `adjust-dateTime-to-timezone( )`
- [2.0] `adjust-time-to-timezone( )`
- [2.0] `day-from-date( )`
- [2.0] `day-from-dateTime( )`
- [2.0] `days-from-duration( )`
- [2.0] `format-date( )`
- [2.0] `format-dateTime( )`
- [2.0] `format-time( )`
- [2.0] `hours-from-dateTime( )`
- [2.0] `hours-from-duration( )`
- [2.0] `hours-from-time( )`
- [2.0] `minutes-from-dateTime( )`
- [2.0] `minutes-from-duration( )`
- [2.0] `minutes-from-time( )`
- [2.0] `month-from-date( )`
- [2.0] `month-from-dateTime( )`
- [2.0] `months-from-duration( )`
- [2.0] `seconds-from-dateTime( )`
- [2.0] `seconds-from-duration( )`
- [2.0] `seconds-from-time( )`
- [2.0] `timezone-from-date( )`
- [2.0] `timezone-from-dateTime( )`
- [2.0] `timezone-from-time( )`
- [2.0] `year-from-date( )`
- [2.0] `year-from-dateTime( )`
- [2.0] `years-from-duration( )`

The [2.0] `format-date( )`, [2.0] `format-dateTime( )`, and [2.0] `format-time( )` functions are defined in XSLT 2.0.

Node Functions

These functions perform operations on nodes:

- `lang( )`
- `local-name( )`
- `name( )`
- `namespace-uri( )`
- `number( )`
- [2.0] `root( )`

Numeric Functions

These functions work with numeric values. Some of them work with a single number, while others work with node-sets or sequences:

- [2.0] `abs( )`
- [2.0] `avg( )`
- `ceiling( )`
- `floor( )`
- `format-number( )`
- [2.0] `max( )`
- [2.0] `min( )`
- `round( )`
- [2.0] `round-half-to-even( )`
- `sum( )`

QName Functions

These functions work with QNames (the XML Schema datatype `xs:QName`):

- [2.0] `in-scope-prefixes( )`
- [2.0] `local-name-from-QName( )`
- [2.0] `namespace-uri-for-prefix( )`
- [2.0] `namespace-uri-from-QName( )`
- [2.0] `prefix-from-QName( )`

- [2.0] `QName( )`
- [2.0] `resolve-QName( )`

Regular Expression Functions

These functions are part of the regular expression support in XSLT 2.0 and XPath 2.0:

- [2.0] `matches( )`
- [2.0] `regex-group( )`
- [2.0] `replace( )`
- [2.0] `tokenize( )`

Sequence or Node-Set Functions

These functions work with *[1.0]* node-sets or *[2.0]* sequences:

- [2.0] `avg( )`
- `boolean( )`
- `count( )`
- [2.0] `deep-equal( )`
- [2.0] `distinct-values( )`
- [2.0] `doc-available( )`
- [2.0] `doc( )`
- [2.0] `empty( )`
- [2.0] `exactly-one( )`
- [2.0] `exists( )`
- [2.0] `index-of( )`
- [2.0] `insert-before( )`
- [2.0] `max( )`
- [2.0] `min( )`
- [2.0] `one-or-more( )`
- [2.0] `remove( )`
- [2.0] `reverse( )`
- [2.0] `subsequence( )`
- `sum( )`
- [2.0] `unordered( )`
- [2.0] `zero-or-one( )`

String Functions

These functions help you manipulate strings:

- [2.0] `codepoint-equal( )`
- [2.0] `codepoints-to-string( )`
- [2.0] `compare( )`
- `concat( )`
- `contains( )`
- [2.0] `encode-for-uri( )`
- [2.0] `ends-with( )`
- [2.0] `escape-html-uri( )`
- [2.0] `iri-to-uri( )`
- [2.0] `lower-case( )`
- [2.0] `matches( )`
- `normalize-space( )`
- [2.0] `normalize-unicode( )`
- [2.0] `replace( )`
- `starts-with( )`
- [2.0] `string-join( )`
- `string-length( )`
- [2.0] `string-to-codepoints( )`
- `substring-after( )`
- `substring-before( )`
- `substring( )`
- [2.0] `tokenize( )`
- `translate( )`
- [2.0] `upper-case( )`

Miscellaneous Functions

Here are functions that don't fit under any other category:

- [2.0] `collection( )`
- [2.0] `doc-available( )`
- [2.0] `doc( )`
- `document( )`
- [2.0] `error( )`

- [2.0] `resolve-uri( )`
- `system-property( )`
- [2.0] `trace( )`
- [2.0] `unparsed-entity-public-id( )`
- `unparsed-entity-uri( )`
- [2.0] `unparsed-text-available( )`
- [2.0] `unparsed-text( )`

Collation Functions

There are several functions that allow you to specify a collation function that compares or sorts text based on the conventions of specific languages. (All of these have been listed previously as string or sequence functions. This subset contains all the functions that work with collations.)

- [2.0] `compare( )`
- `contains( )`
- [2.0] `deep-equal( )`
- [2.0] `distinct-values( )`
- [2.0] `index-of( )`
- `substring-after( )`
- `substring-before( )`

[2.0] abs()

Returns the absolute value of a numeric argument.

Syntax

```
numeric? abs(numeric?)
```

Input

A numeric value.

Output

The absolute value of the given numeric value.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 6.4, "Functions on Numeric Values."

Example

Here's a short stylesheet that tests the `abs( )` function:

```
<?xml version="1.0"?>
<!-- abs.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Here are some tests of the abs() function:&#xA;</xsl:text>
    <xsl:text>&#xA;  abs(7) = </xsl:text>
    <xsl:value-of select="abs(7)"/>
    <xsl:text>&#xA;  abs(-7) = </xsl:text>
    <xsl:value-of select="abs(-7)"/>
    <xsl:text>&#xA;  abs(0) = </xsl:text>
    <xsl:value-of select="abs(0)"/>
    <xsl:text>&#xA;  abs(-0) = </xsl:text>
    <xsl:value-of select="abs(-0)"/>

    <!-- An XSLT 2.0 processor won't run this example at all. -->
    <!-- <xsl:value-of select="abs('x')"/> -->

    <xsl:variable name="testSequence" as="xs:integer*" select="1 to 10"/>
    <xsl:text>&#xA;  $testSequence = </xsl:text>
    <xsl:value-of select="$testSequence" separator=", "/>
    <xsl:text>&#xA;  abs(count($testSequence)) = </xsl:text>
    <xsl:value-of select="abs(count($testSequence))"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Here are some tests of the abs() function:

  abs(7) = 7
  abs(-7) = 7
  abs(0) = 0
  abs(-0) = 0
  $testSequence = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
  abs(count($testSequence)) = 10
```

The last call to `abs( )` uses the `count( )` function to create a numeric value. As we mentioned in a comment in the stylesheet, any attempt to pass a nonnumeric value to the `abs( )` function is a static error. You can uncomment the line `<xsl:value-of select="abs('x')/>` and try it yourself if you like.

[2.0] adjust-date-to-timezone()

Adjusts an `xs:date` value to a particular timezone.

Syntax

```
xs:date? adjust-date-to-timezone(xs:date?)
xs:date? adjust-date-to-timezone(xs:date?, $timezone as xs:dayTimeDuration)
```

Input

An optional `xs:date` value and an optional `xs:dayTimeDuration`. If no `xs:date` is provided, the empty sequence is returned. If an `xs:dayTimeDuration` is provided, the date is adjusted to the timezone it contains; otherwise, the XSLT processor uses the default timezone as returned by [2.0] `implicit-timezone( )`. Finally, if the timezone provided by the `xs:dayTimeDuration` is the empty sequence, the function returns the date with the timezone information removed.

Output

An `xs:date` value adjusted to the appropriate timezone.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.7, "Timezone Adjustment Functions on Dates and Time Values."

Example

Here's a stylesheet that uses the `adjust-date-to-timezone( )` in several ways:

```
<?xml version="1.0"?>
<!-- adjust-date-to-timezone.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="gmt" select="xs:dayTimeDuration('PT0H')"/>
    <xsl:variable name="est" select="xs:dayTimeDuration('-PT5H')"/>
    <xsl:variable name="cst" select="xs:dayTimeDuration('-PT6H')"/>
    <xsl:variable name="minusTen" select="xs:dayTimeDuration('-PT10H')"/>

    <xsl:variable name="LilysBirthday" as="xs:date"
      select="xs:date('1995-04-21')"/>
    <xsl:variable name="format"
      select="'[FNn], [MNn] [D1], [Y0001]'"/>

    <xsl:text>&#xA;Here are some tests of the </xsl:text>
    <xsl:text>adjust-date-to-timezone() function:&#xA;</xsl:text>

    <xsl:text>&#xA;  My daughter was born on &#x9;</xsl:text>
    <xsl:value-of
      select="format-date($LilysBirthday, $format)"/>
    <xsl:text>&#xA;    adjusted to GMT: &#x9;&#x9;</xsl:text>
    <xsl:value-of
      select="format-date(
              adjust-date-to-timezone($LilysBirthday, $gmt),
              $format)"/>
```

```
    <xsl:text>&#xA;    adjusted to EST: &#x9;&#x9;</xsl:text>
    <xsl:value-of
      select="format-date
              (adjust-date-to-timezone($LilysBirthday, $est),
              $format)"/>

    <xsl:text>&#xA;    adjusted to CST: &#x9;&#x9;</xsl:text>
    <xsl:value-of
      select="format-date
              (adjust-date-to-timezone($LilysBirthday, $cst),
              $format)"/>

    <xsl:text>&#xA;&#xA;  </xsl:text>
    <xsl:text>The current date in the default timezone is: </xsl:text>
    <xsl:text>&#xA;&#x9;&#x9;&#x9;&#x9;</xsl:text>
    <xsl:value-of
      select="format-date(
              adjust-date-to-timezone(current-date()),
              $format)"/>

    <xsl:text>&#xA;    adjusted to GMT: &#x9;&#x9;</xsl:text>
    <xsl:value-of
      select="format-date
              (adjust-date-to-timezone(current-date(), $gmt),
              $format)"/>

    <xsl:text>&#xA;    adjusted to GMT-10: &#x9;</xsl:text>
    <xsl:value-of
      select="format-date(
              adjust-date-to-timezone(current-date(), $minusTen),
              $format)"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results from this stylesheet:

```
Here are some tests of the adjust-date-to-timezone() function:

  My daughter was born on       Friday, April 21, 1995
    adjusted to GMT:            Friday, April 21, 1995
    adjusted to EST:            Friday, April 21, 1995
    adjusted to CST:            Friday, April 21, 1995

  The current date in the default timezone is:
                                Thursday, November 16, 2007
    adjusted to GMT:            Thursday, November 16, 2007
    adjusted to GMT-10:         Wednesday, November 15, 2007
```

Notice that we used the `format-date( )` function to print the `xs:date` values.

See Also

The definitions of the [2.0] `adjust-dateTime-to-timezone( )`, [2.0] `adjust-time-to-timezone( )`, and [2.0] `format-date( )` functions.

[2.0] adjust-dateTime-to-timezone()

Adjusts an `xs:dateTime` value to a particular timezone.

Syntax

```
xs:dateTime? adjust-dateTime-to-timezone(xs:dateTime?)
xs:dateTime? adjust-dateTime-to-timezone(xs:dateTime?,
                                         $timezone as xs:dayTimeDuration?)
```

Inputs

An optional `xs:dateTime` value and an optional `xs:dayTimeDuration`. If no `xs:dateTime` is provided, the empty sequence is returned. If an `xs:dayTimeDuration` is supplied, the `xs:dateTime` value is adjusted to the timezone it contains. Otherwise, `adjust-dateTime-to-timezone( )` adjusts the `xs:dateTime` value to the default timezone as returned by [2.0] `implicit-timezone( )`. Finally, if the timezone provided is the empty sequence, the function returns the `xs:dateTime` with the timezone information removed.

Output

The given `xs:dateTime` value adjusted to the appropriate timezone.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.7, "Timezone Adjustment Functions on Dates and Time Values."

Example

This stylesheet uses `adjust-dateTime-to-timezone( )` in several different ways. The default timezone here is GMT -4, representing Eastern Daylight Time (EDT).

```
<?xml version="1.0"?>
<!-- adjust-datetime-to-timezone.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="gmt" select="xs:dayTimeDuration('PT0H')"/>
    <xsl:variable name="est" select="xs:dayTimeDuration('-PT5H')"/>
    <xsl:variable name="cst" select="xs:dayTimeDuration('-PT6H')"/>
    <xsl:variable name="minusTen" select="xs:dayTimeDuration('-PT10H')"/>

    <xsl:variable name="LilysBirthday" as="xs:dateTime"
      select="xs:dateTime('1995-04-21T00:43:00-05:00')"/>
    <xsl:variable name="format"
      select="'[FNn], [MNn] [D1], [Y0001], at [h1]:[m01]:[s01]'"/>

    <xsl:text>&#xA;Here are some tests of the </xsl:text>
    <xsl:text>adjust-dateTime-to-timezone() function:&#xA;</xsl:text>
```

```
    <xsl:text>&#xA;  My daughter was born on &#x9;</xsl:text>
    <xsl:value-of
      select="format-dateTime($LilysBirthday, $format)"/>
    <xsl:text>&#xA;    adjusted to GMT: &#x9;&#x9;</xsl:text>
    <xsl:value-of
      select="format-dateTime(
              adjust-dateTime-to-timezone($LilysBirthday, $gmt),
              $format)"/>

    <xsl:text>&#xA;    adjusted to EST: &#x9;&#x9;</xsl:text>
    <xsl:value-of
      select="format-dateTime
              (adjust-dateTime-to-timezone($LilysBirthday, $est),
              $format)"/>

    <xsl:text>&#xA;    adjusted to CST: &#x9;&#x9;</xsl:text>
    <xsl:value-of
      select="format-dateTime
              (adjust-dateTime-to-timezone($LilysBirthday, $cst),
              $format)"/>

    <xsl:text>&#xA;&#xA;  </xsl:text>
    <xsl:text>The current time in the default timezone is: </xsl:text>
    <xsl:text>&#xA;  &#x9;&#x9;&#x9;&#x9;</xsl:text>
    <xsl:value-of
      select="format-dateTime(
              adjust-dateTime-to-timezone(current-dateTime()),
              $format)"/>

    <xsl:text>&#xA;    adjusted to GMT: &#x9;&#x9;</xsl:text>
    <xsl:value-of
      select="format-dateTime
              (adjust-dateTime-to-timezone(current-dateTime(), $gmt),
              $format)"/>

    <xsl:text>&#xA;    adjusted to GMT-10: &#x9;</xsl:text>
    <xsl:value-of
      select="format-dateTime(
              adjust-dateTime-to-timezone(current-dateTime(), $minusTen),
              $format)"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results of our stylesheet:

```
Here are some tests of the adjust-dateTime-to-timezone() function:

  My daughter was born on       Friday, April 21, 1995, at 12:43:00
    adjusted to GMT:            Friday, April 21, 1995, at 5:43:00
    adjusted to EST:            Friday, April 21, 1995, at 12:43:00
    adjusted to CST:            Thursday, April 20, 1995, at 11:43:00

  The current time in the default timezone is:
                                Thursday, November 16, 2006, at 3:34:29
```

```
adjusted to GMT:            Thursday, November 16, 2006, at 8:34:29
adjusted to GMT-10:         Wednesday, November 15, 2006, at 10:34:29
```

The default timezone is considered part of the context; you can invoke the function `implicit-timezone( )` to get the default. See the definition of the [2.0] `implicit-timezone( )` function for more information. Also notice that we used the `format-dateTime( )` function to format the `xs:dateTime` values.

See Also

The definitions of the [2.0] `adjust-date-to-timezone( )`, [2.0] `adjust-time-to-timezone( )`, and [2.0] `format-dateTime( )` functions.

[2.0] adjust-time-to-timezone()

Adjusts an `xs:time` value to a particular timezone.

Syntax

```
xs:time? adjust-time-to-timezone(xs:time?)
xs:time? adjust-time-to-timezone(xs:time?, $timezone as xs:dayTimeDuration?)
```

Input

An optional `xs:time` value and an optional `xs:dayTimeDuration`. If no `xs:date` is provided, the empty sequence is returned. If an `xs:dayTimeDuration` is provided, the `xs:time` value is adjusted to the timezone contained in the `xs:dayTimeDuration`; otherwise, the XSLT processor uses the default timezone as returned by [2.0] `implicit-timezone( )`. Finally, if the timezone provided by the `xs:dayTimeDuration` is the empty sequence, the function returns the `xs:time` with the timezone information removed.

Output

The given `xs:time` value adjusted to the appropriate timezone.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.7, "Timezone Adjustment Functions on Dates and Time Values."

Example

The following stylesheet tests the `adjust-time-to-timezone( )` function:

```
<?xml version="1.0"?>
<!-- adjust-time-to-timezone.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
```

```
  <xsl:variable name="gmt" select="xs:dayTimeDuration('PT0H')"/>
  <xsl:variable name="est" select="xs:dayTimeDuration('-PT5H')"/>
  <xsl:variable name="cst" select="xs:dayTimeDuration('-PT6H')"/>
  <xsl:variable name="minusTen" select="xs:dayTimeDuration('-PT10H')"/>

  <xsl:variable name="LilysBirthday" as="xs:time"
    select="xs:time('00:43:00-05:00')"/>
  <xsl:variable name="format"
    select="'[h1]:[m01]:[s01] [P]'"/>

  <xsl:text>&#xA;Here are some tests of the </xsl:text>
  <xsl:text>adjust-time-to-timezone() function:&#xA;</xsl:text>

  <xsl:text>&#xA;  My daughter was born at &#x9;</xsl:text>
  <xsl:value-of
    select="format-time($LilysBirthday, $format)"/>
  <xsl:text>&#xA;    adjusted to GMT: &#x9;&#x9;</xsl:text>
  <xsl:value-of
    select="format-time(
            adjust-time-to-timezone($LilysBirthday, $gmt),
            $format)"/>

  <xsl:text>&#xA;    adjusted to EST: &#x9;&#x9;</xsl:text>
  <xsl:value-of
    select="format-time
            (adjust-time-to-timezone($LilysBirthday, $est),
            $format)"/>

  <xsl:text>&#xA;    adjusted to CST: &#x9;&#x9;</xsl:text>
  <xsl:value-of
    select="format-time
            (adjust-time-to-timezone($LilysBirthday, $cst),
            $format)"/>

  <xsl:text>&#xA;&#xA;  </xsl:text>
  <xsl:text>The current time in the default timezone is: </xsl:text>
  <xsl:text>&#xA;  &#x9;&#x9;&#x9;&#x9;</xsl:text>
  <xsl:value-of
    select="format-time(
            adjust-time-to-timezone(current-time()),
            $format)"/>

  <xsl:text>&#xA;    adjusted to GMT: &#x9;&#x9;</xsl:text>
  <xsl:value-of
    select="format-time
            (adjust-time-to-timezone(current-time(), $gmt),
            $format)"/>

  <xsl:text>&#xA;    adjusted to GMT-10: &#x9;</xsl:text>
  <xsl:value-of
    select="format-time(
            adjust-time-to-timezone(current-time(), $minusTen),
            $format)"/>
</xsl:template>
```

```
</xsl:stylesheet>
```

Here are the stylesheet results:

```
Here are some tests of the adjust-time-to-timezone() function:

  My daughter was born at       12:43:00 a.m.
    adjusted to GMT:            5:43:00 a.m.
    adjusted to EST:            12:43:00 a.m.
    adjusted to CST:            11:43:00 p.m.

  The current time in the default timezone is:
                                3:35:44 a.m.
    adjusted to GMT:            8:35:44 a.m.
    adjusted to GMT-10:         10:35:44 p.m.
```

Notice that we used the `format-time( )` function to print the `xs:time` values.

See Also

The definitions of the [2.0] `adjust-date-to-timezone( )`, [2.0] `adjust-dateTime-to-timezone( )`, and [2.0] `format-time( )` functions.

[2.0] avg()

Given a sequence, returns the average value of the items in the sequence.

Syntax

```
xs:anyAtomicType? avg(xs:anyAtomicType*)
```

Input

A sequence of values.

Output

The average of the given sequence. You can calculate averages for six different datatypes: `xs:integer`, `xs:double`, `xs:decimal`, `xs:float`, `xs:yearMonthDuration`, and `xs:dayTimeDuration`.

Given a sequence of numeric values, the XSLT processor returns the average of those numbers, converting datatypes as necessary. Given a sequence of durations, the XSLT processor returns the average of those durations.

The `avg( )` function assumes you'll send it a sequence containing sensible data; if not, the XSLT processor throws an error. Asking for the average of the sequence `(42, 57, 'blue')` returns an error, as you'd expect.

Some notes about how the `avg( )` function works:

- To calculate the average of a sequence of durations, all the values must be `xs:dayTimeDuration`s or `xs:yearMonthDuration`s. You can't mix the two types of durations; if you do, the XSLT processor throws an error.
- If all the items in the sequence are of type `xs:untypedAtomic`, the XSLT processor attempts to cast each value to `xs:double`. If *any* item in the sequence can't be converted to an `xs:double`, the XSLT processor throws an error.
- Finally, if you pass the `avg( )` function the empty sequence, the function returns the empty sequence. Although you're not giving the function any useful data in this case, the XSLT processor doesn't throw an error.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.4, "Aggregate Functions."

Example

Here's a stylesheet that calculates the averages of several sequences:

```
<?xml version="1.0"?>
<!-- avg.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="seq1" select="(3, 5, 18)"/>
    <xsl:variable name="seq2" select="(3, 5, 48.273, 2.9e3)"/>

    <xsl:variable name="value1" as="xs:integer" select="42"/>
    <xsl:variable name="value2" as="xs:double" select="2718.28E-3"/>
    <xsl:variable name="value3" as="xs:float" select="98.6"/>
    <xsl:variable name="value4" as="xs:decimal" select="2.54"/>
    <xsl:variable name="seq3"
      select="($value1, $value2, $value3, $value4)"/>

    <xsl:variable name="seq4"
      select="(xs:yearMonthDuration('P3Y8M'),
               xs:yearMonthDuration('P4Y2M'),
               xs:yearMonthDuration('P6Y4M'))"/>
    <xsl:variable name="seq5"
      select="(xs:dayTimeDuration('P2DT4H23M12.2S'),
               xs:dayTimeDuration('P3DT8H17M'),
               xs:dayTimeDuration('P3D'))"/>

    <xsl:text>&#xA;Here are some tests of the avg() function:&#xA;</xsl:text>

    <xsl:text>&#xA;  avg(</xsl:text>
    <xsl:value-of select="$seq1" separator=", "/>
    <xsl:text>) = </xsl:text>
    <xsl:value-of select="format-number(avg($seq1), '#.###')"/>
```

```
<xsl:text>&#xA;&#xA;  avg(</xsl:text>
<xsl:value-of select="$seq2" separator=", "/>
<xsl:text>) = </xsl:text>
<xsl:value-of select="format-number(avg($seq2), '#.###')"/>

<xsl:text>&#xA;&#xA;  avg(</xsl:text>
<xsl:value-of select="$seq3" separator=", "/>
<xsl:text>) = </xsl:text>
<xsl:value-of select="format-number(avg($seq3), '#.###')"/>

<xsl:text>&#xA;&#xA;  avg(</xsl:text>
<xsl:value-of select="$seq4" separator=", "/>
<xsl:text>) = </xsl:text>
<xsl:value-of select="avg($seq4)"/>

<xsl:text>&#xA;&#xA;    In text, the average of</xsl:text>
<xsl:for-each select="$seq4">
  <xsl:text>&#xA;      </xsl:text>
  <xsl:value-of select="years-from-duration(.)"/>
  <xsl:text> years and </xsl:text>
  <xsl:value-of select="months-from-duration(.)"/>
  <xsl:text> months (</xsl:text>
  <xsl:value-of select="."/>
  <xsl:text>)</xsl:text>
</xsl:for-each>

<xsl:text>&#xA;    is </xsl:text>
<xsl:value-of select="years-from-duration(avg($seq4))"/>
<xsl:text> years and </xsl:text>
<xsl:value-of select="months-from-duration(avg($seq4))"/>
<xsl:text> months (</xsl:text>
<xsl:value-of select="avg($seq4)"/>
<xsl:text>).</xsl:text>

<xsl:text>&#xA;&#xA;  avg(</xsl:text>
<xsl:value-of select="$seq5" separator=", "/>
<xsl:text>) = </xsl:text>
<xsl:variable name="avg5" select="avg($seq5)"/>
<xsl:value-of select="$avg5"/>

<xsl:text>&#xA;&#xA;    In text, the average of</xsl:text>
<xsl:for-each select="$seq5">
  <xsl:text>&#xA;      </xsl:text>
  <xsl:value-of select="days-from-duration(.)"/>
  <xsl:text> days, </xsl:text>
  <xsl:value-of select="hours-from-duration(.)"/>
  <xsl:text> hours, </xsl:text>
  <xsl:value-of select="minutes-from-duration(.)"/>
  <xsl:text> minutes and </xsl:text>
  <xsl:value-of
    select="format-number(seconds-from-duration(.), '#.##')"/>
  <xsl:text> seconds (</xsl:text>
  <xsl:value-of select="."/>
  <xsl:text>)</xsl:text>
</xsl:for-each>
```

```
      <xsl:text>&#xA;    is </xsl:text>
        <xsl:value-of select="days-from-duration($avg5)"/>
        <xsl:text> days, </xsl:text>
        <xsl:value-of select="hours-from-duration($avg5)"/>
        <xsl:text> hours, </xsl:text>
        <xsl:value-of select="minutes-from-duration($avg5)"/>
        <xsl:text> minutes and </xsl:text>
        <xsl:value-of
          select="format-number(seconds-from-duration($avg5), '#.##')"/>
        <xsl:text> seconds.</xsl:text>

    </xsl:template>

  </xsl:stylesheet>
```

Here are the results from this stylesheet:

```
  Here are some tests of the avg() function:

    avg(3, 5, 18) = 8.667

    avg(3, 5, 48.273, 2900) = 739.068

    avg(42, 2.71828, 98.6, 2.54) = 36.465

    avg(P3Y8M, P4Y2M, P6Y4M) = P4Y9M

      In text, the average of
        3 years and 8 months (P3Y8M)
        4 years and 2 months (P4Y2M)
        6 years and 4 months (P6Y4M)
      is 4 years and 9 months (P4Y9M).

    avg(P2DT4H23M12.2S, P3DT8H17M, P3D) = P2DT2OH13M24.066666S

      In text, the average of
        2 days, 4 hours, 23 minutes and 12.2 seconds (P2DT4H23M12.2S)
        3 days, 8 hours, 17 minutes and 0 seconds (P3DT8H17M)
        3 days, 0 hours, 0 minutes and 0 seconds (P3D)
      is 2 days, 20 hours, 13 minutes and 24.07 seconds.
```

The stylesheet demonstrates five different types of sequences. The first is all integers, and the second is a sequence of numbers. All the values in the first two sequences can be converted to numbers, so the results are what we expect. The third sequence features four numeric values, each of which is a different numeric type (`xs:integer`, `xs:decimal`, `xs:float`, and `xs:double`). The last two sequences are made up of the two types of durations. The stylesheet also uses functions such as `years-from-duration( )`, and `format-number( )` to format the data.

The sequence `(3, 5, '18')` won't work because one of the items in the sequence is a string. Although the string `'18'` could obviously be converted to a number with the `number( )` function, the `avg( )` function puts the burden on us to make sure all the literal values in the sequence are numbers or durations. However, if the sequence contains untyped values (nodes read in from a document that doesn't use a schema, for example), the XSLT processor attempts to

cast all of those values to `xs:double`. If any of them can't be cast to `xs:double`, the XSLT processor raises an error.

A final note: if you want to store a numeric average in a typed variable, the `xs:double` datatype is the most flexible. Using the most restrictive numeric type, `xs:integer`, is a really bad idea. For example, the XSLT processor binds this variable without complaint:

```
<xsl:variable name="avg1" as="xs:integer" select="avg((3, 5, 4))"/>
```

On the other hand, this generates a static error:

```
<xsl:variable name="avg2" as="xs:integer" select="avg((3, 5, 18))"/>
```

If we use `avg( )` to set the value of an integer variable dynamically, the XSLT processor throws an error if the returned value isn't an `xs:integer`. We could get this to work by casting the result to an `xs:integer`:

```
<xsl:variable name="avg2" as="xs:integer"
  select="xs:integer(avg((3, 5, 18)))"/>
```

[2.0] base-uri()

Returns the base URI of a given node.

Syntax

```
xs:anyURI? base-uri(node()?)
xs:anyURI? base-uri()
```

Input

A node. Without an argument, `base-uri( )` returns the base URI of the context item.

Output

The base URI of the given node.

The `base-uri( )` function works with the XML Base specification. By default the base URI is the URI of the XML document itself, but a document can change that by using the `xml:base` attribute on any element.

If a given node does not have a base URI property and the node has a parent, `base-uri( )` looks at the node's ancestors. The function attempts to find the base URI of the node's parent, then its parent's parent, and so forth, until it either finds a base URI property or reaches a node that does not have a parent. If no base URI property can be found, `base-uri( )` returns the empty sequence. If the argument to `base-uri( )` is the empty sequence, the function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 2, "Accessors."

Example

Here's a revised version of our list of cars. Notice that the `<manufacturer>` elements all use the `xml:base` attribute to define a new base URI:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- xmlbase.xml -->
<cars>
  <manufacturer name="Chevrolet"
    xml:base="http://www.chevrolet.com/">
    <car>Cavalier</car>
    <car>Corvette</car>
    <car>Impala</car>
    <car>Malibu</car>
  </manufacturer>
  <manufacturer name="Ford"
    xml:base="http://www.ford.com/">
    <car>Pinto</car>
    <car>Mustang</car>
    <car>Taurus</car>
  </manufacturer>
  <manufacturer name="Volkswagen"
    xml:base="http://www.vw.com/">
    <car>Beetle</car>
    <car>Jetta</car>
    <car>Passat</car>
    <car>Touraeg</car>
  </manufacturer>
</cars>
```

We'll test the `base-uri( )` function with this stylesheet:

```
<?xml version="1.0"?>
<!-- base-uri.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the base-uri() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  The base URI for the </xsl:text>
    <xsl:text>document root is:&#xA;    </xsl:text>
    <xsl:value-of select="base-uri()"/>

    <xsl:text>&#xA;&#xA;  The base URI for the </xsl:text>
    <xsl:text>&lt;cars&gt; element is:&#xA;    </xsl:text>
    <xsl:value-of select="base-uri(cars)"/>

    <xsl:for-each select="/cars/manufacturer">
      <xsl:text>&#xA;&#xA;  The base URI for the </xsl:text>
      <xsl:text>manufacturer named </xsl:text>
      <xsl:value-of select="@name"/>
      <xsl:text>: &#xA;    </xsl:text>
      <xsl:value-of select="base-uri()"/>
```

```
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

When running this stylesheet against our modified list of cars, we get these results:

```
Tests of the base-uri() function:

  The base URI for the document root is:
    file:/C:/projects/XSLTbookV2/AppendixC/xmlbase.xml

  The base URI for the <cars> element is:
    file:/C:/projects/XSLTbookV2/AppendixC/xmlbase.xml

  The base URI for the manufacturer named Chevrolet:
    http://www.chevrolet.com/

  The base URI for the manufacturer named Ford:
    http://www.ford.com/

  The base URI for the manufacturer named Volkswagen:
    http://www.vw.com/
```

The `base-uri( )` function would be very useful if we were generating links from this XML document. Here's a stylesheet that creates links for the `<car>` elements using the `base-uri( )` function:

```
<?xml version="1.0"?>
<!-- base-uri2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" include-content-type="no"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Here are some cars</title>
      </head>
      <body>
        <h1>Here are some cars:</h1>
        <xsl:apply-templates select="cars/manufacturer"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="manufacturer">
    <h2>
      <xsl:value-of select="@name"/>
    </h2>
    <ul>
      <xsl:for-each select="car">
        <li>
          <a href="{concat(base-uri(), ., '.html')}">
            <xsl:value-of select="."/>
```

```
            </a>
          </li>
        </xsl:for-each>
      </ul>
    </xsl:template>

</xsl:stylesheet>
```

To generate the links, we use an attribute value template and the `concat( )` function to combine the value of the base URI, the value of the current element (the dot represents the `<car>` we're currently processing), and the string `.html`. This generates a series of links that look like this:

```
<h2>Volkswagen</h2>
<ul>
   <li><a href="http://www.vw.com/Beetle.html">Beetle</a></li>
   <li><a href="http://www.vw.com/Jetta.html">Jetta</a></li>
   <li><a href="http://www.vw.com/Passat.html">Passat</a></li>
   <li><a href="http://www.vw.com/Touraeg.html">Touraeg</a></li>
</ul>
```

Our assumption here is that the base URI property ends with a slash, as all the `xml:base` attributes in our source document did. More robust code would check the value returned by `base-uri( )` and concatenate a slash if necessary.

boolean()

Converts its argument to a boolean value.

Syntax

```
[1.0] boolean boolean(object)
[2.0] xs:boolean boolean(item()*)
```

Inputs

An object. The object is converted to a boolean value. This conversion is described in the following subsection.

Output

[1.0] The boolean value corresponding to the input object. Objects are converted to boolean values as follows:

- A number is `true` if and only if it is not zero, negative zero, or `NaN` (not a number).
- A node-set is `true` if and only if it is not empty.
- A string is `true` if and only if its length is greater than zero.
- All other datatypes are converted in a way specific to those datatypes.

[2.0] For XSLT 2.0, things are more complicated. The value returned, as you'd expect, is an `xs:boolean`. Here's how the argument to `boolean( )` is converted to a boolean value:

- If the argument is a singleton of any numeric type, `boolean( )` returns `false` if the value is zero or `NaN` (not a number); everything else returns `true`.
- If the argument is a singleton of type `xs:string`, `xs:anyURI`, `xs:untypedAtomic`, or any type derived from them, `boolean( )` returns `true` if the argument has a length greater than zero.
- If the argument is a singleton of type `xs:boolean` (or of a type derived from `xs:boolean`), `boolean( )` simply returns the argument as is.
- If the argument is a sequence whose first item is a node, `boolean( )` returns `true`.
- If the argument is the empty sequence, `boolean( )` returns `false`.
- If the argument is anything else (`xs:date`, `xs:time`, or a sequence of multiple atomic values, for example), `boolean( )` raises an error.

Defined in

[1.0] XPath section 4.3, "Boolean Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 15.1, "General Functions and Operators on Sequences."

Example

We'll use our sales document to illustrate the `boolean( )` function:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

Here's our stylesheet:

```
<?xml version="1.0"?>
<!-- boolean.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;</xsl:text>
    <xsl:text>Tests of the boolean() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  boolean(true()) = </xsl:text>
    <xsl:value-of select="boolean(true())"/>

    <xsl:text>&#xA;  boolean(true) = </xsl:text>
    <xsl:value-of select="boolean(true)"/>

    <xsl:text>&#xA;  boolean('false') = </xsl:text>
    <xsl:value-of select="boolean('false')"/>

    <xsl:text>&#xA;  boolean('7') = </xsl:text>
    <xsl:value-of select="boolean('7')"/>

    <xsl:text>&#xA;  boolean(7) = </xsl:text>
    <xsl:value-of select="boolean(7)"/>

    <xsl:text>&#xA;  boolean(/report/brand/units[. > 20000]) = </xsl:text>
    <xsl:value-of select="boolean(/report/brand/units[. > 20000])"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Tests of the boolean() function:

  boolean(true()) = true
  boolean(true) = false
  boolean('false') = true
  boolean('7') = true
  boolean(7) = true
  boolean(/report/brand/units[. > 20000]) = true
```

For the first test, the argument is a call to the `true( )` function, which always returns `true`. Because this is a boolean value already, the `boolean( )` function simply returns that value as is. For the second test, we're asking for all the `<true>` elements in the current context; there are no `<true>` elements, so this is `false`. The argument of the third and fourth tests are strings whose length is greater than zero, so `boolean( )` returns `true`. The fifth argument is a number greater than zero, so it is `true` as well.

The final example uses an XPath statement to select all of the `<units>` elements in our sales report that have a numeric value greater than `20000`. Because this selects at least one node, `boolean( )` returns `true`. This works in XSLT 1.0 because it generates a node-set with at least one member, and it works in XSLT 2.0 because it creates a sequence whose first member is a

node. If we changed the expression to look for sales figures greater than `30000`, `boolean( )` would return `false`.

Here's a stylesheet that uses XSLT 2.0's rules to process various sequences:

```
<?xml version="1.0"?>
<!-- boolean2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;</xsl:text>
    <xsl:text>Tests of the boolean() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  boolean(()) = </xsl:text>
    <xsl:value-of select="boolean(())"/>

    <xsl:variable name="testSequence1" as="item()*">
      <xsl:sequence select="(3)"/>
    </xsl:variable>

    <xsl:text>&#xA;&#xA;  $testSequence1 = (</xsl:text>
    <xsl:value-of select="$testSequence1" separator=", "/>
    <xsl:text>)&#xA;  boolean($testSequence1) = </xsl:text>
    <xsl:value-of select="boolean($testSequence1)"/>

    <xsl:variable name="testSequence2" as="item()*">
      <xsl:sequence select="/report/brand/units"/>
      <xsl:sequence select="(3, 4, 5)"/>
    </xsl:variable>

    <xsl:text>&#xA;&#xA;  $testSequence2 = (</xsl:text>
    <xsl:value-of select="$testSequence2" separator=", "/>
    <xsl:text>)&#xA;  boolean($testSequence2) = </xsl:text>
    <xsl:value-of select="boolean($testSequence2)"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Tests of the boolean() function:

  boolean(()) = false

  $testSequence1 = (3)
  boolean($testSequence1) = true

  $testSequence2 = (27408, 8203, 22101, 14336, 19268, 3, 4, 5)
  boolean($testSequence2) = true
```

There are three different tests here. For the first test, we pass the `boolean( )` function the empty sequence, so the result is `false`. In the second test, we pass a singleton numeric value to `boolean( )`; because that value is nonzero, the result is `true`. In the final test, the argument we

use is a sequence that contains five <units> nodes, followed by three numeric values. Because the first item in the sequence is a node, boolean( ) returns true.

In this stylesheet, if we define $testSequence1 to have more than one item, the stylesheet would fail to compile. Any sequence we send to boolean( ) must be a singleton or a sequence whose first value is a node. For the second sequence, $testSequence2, we can add as many items of as many different types as we like, as long as the first item is a node. If there were no nodes that matched the expression /report/brand/units, the first node in the sequence would be the atomic value 3, so the stylesheet would generate a runtime error.

See "Converting to boolean values" in Chapter 5 for more examples and information.

ceiling()

Returns the smallest integer that is not less than the argument.

Syntax

```
[1.0] number? ceiling(number?)
[2.0] numeric? ceiling(numeric?)
```

Inputs

A number.

[1.0] If the argument is not a number, it is transformed into a number as if it had been processed by the number( ) function. If the argument cannot be transformed into a number, the ceiling( ) function returns the value NaN (not a number).

[2.0] In XSLT 2.0, the argument must be one of the four numeric types (xs:float, xs:decimal, xs:double, or xs:integer). If it is not, the XSLT processor raises an error. The result of the ceiling( ) function will be of the same type as the argument.

Output

The smallest integer that is not less than the argument.

[1.0] In XSLT 1.0, ceiling( ) returns NaN if the argument cannot be converted to a number.

[2.0] In XSLT 2.0, ceiling( ) raises an error if the argument cannot be converted to a number.

Defined in

[1.0] XPath section 4.4, "Number Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 6.4, "Functions on Numeric Values."

Example

The following stylesheet shows the results of invoking the ceiling( ) function against a variety of values. We'll use this XML document as input:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
```

```
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

Here's the stylesheet that uses the `ceiling( )` function:

```
<?xml version="1.0"?>
<!-- ceiling.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the ceiling() function:&#xA;&#xA;</xsl:text>

    <xsl:text>  ceiling(7.983) = </xsl:text>
    <xsl:value-of select="ceiling(7.983)"/>

    <xsl:text>&#xA;  ceiling(-7.893) = </xsl:text>
    <xsl:value-of select="ceiling(-7.893)"/>

    <xsl:text>&#xA;  ceiling(avg(/report/brand/units)) = </xsl:text>
    <xsl:value-of select="ceiling(avg(/report/brand/units))"/>

    <xsl:text>&#xA;  ceiling('blue') = </xsl:text>
    <xsl:value-of version="1.0" select="ceiling('blue')"/>

  </xsl:template>

</xsl:stylesheet>
```

When we transform the XML document with our stylesheet, here are the results:

```
Tests of the ceiling() function:

  ceiling(7.983) = 8
```

```
ceiling(-7.893) = -7
ceiling(avg(/report/brand/units)) = 18264
ceiling('blue') = NaN
```

For the last test of the `ceiling( )` function, we specified `version="1.0"` on the `<xsl:value-of>` element. In XSLT 1.0 mode, we get the result `NaN` (not a number). If we process this `<xsl:value-of>` element in XSLT 2.0 mode, the stylesheet won't run at all.

[2.0] codepoint-equal()

Determines whether two strings are equal, based on their Unicode codepoints.

Syntax

```
xs:boolean? codepoint-equal(xs:string?, xs:string?)
```

Inputs

Two `xs:string`s to be compared.

Output

This function returns `true` or `false` depending on whether the two strings are equal using the Unicode code point collation. This function allows you to compare `xs:anyURI` values without having to specify a collation. If either argument is the empty sequence, the result is the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 7.3, "Equality and Comparison of Strings."

Example

Here is a short stylesheet that tests strings to see whether they are equal:

```
<?xml version="1.0"?>
<!-- codepoint-equal.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the codepoint-equal() function:&#xA;</xsl:text>

    <xsl:text>&#xA;  codepoint-equal('A', 'A') = </xsl:text>
    <xsl:value-of select="codepoint-equal('A', 'A')"/>

    <xsl:text>&#xA;  codepoint-equal('A', '&amp;#x41;') = </xsl:text>
    <xsl:value-of select="codepoint-equal('A', '&#x41;')"/>

    <xsl:text>&#xA;  codepoint-equal('A', '&amp;#65;') = </xsl:text>
```

```
    <xsl:value-of select="codepoint-equal('A', '&#65;')"/>

    <xsl:text>&#xA;  codepoint-equal('Strasse', 'Stra&amp;#xDF;e') = </xsl:text>
    <xsl:value-of select="codepoint-equal('Strasse', 'Stra&#xDF;e')"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the stylesheet results:

```
Tests of the codepoint-equal() function:

  codepoint-equal('A', 'A') = true
  codepoint-equal('A', '&#x41;') = true
  codepoint-equal('A', '&#65;') = true
  codepoint-equal('Strasse', 'Straße') = false
```

The first three tests compare the capital letter *A* with itself and with the hexadecimal and decimal representations of itself. The last test compares two spellings of the German word *Strasse*, one of which uses *ss* and one of which uses the German "sharp-s" character (ß). In the Unicode code point collation, these two strings are not equal.

There are some functions in XPath 2.0 and XSLT 2.0 that allow us to specify different collation algorithms (one in which `ss` and ß are equal, for example). See the definitions of the functions [2.0] `compare( )`, `contains( )`, [2.0] `deep-equal( )`, [2.0] `distinct-values( )`, [2.0] `ends-with( )`, [2.0] `index-of( )`, [2.0] `max( )`, [2.0] `min( )`, `starts-with( )`, `substring-after( )`, and `substring-before( )` for examples.

[2.0] codepoints-to-string()

Converts a sequence of Unicode codepoints to a string.

Syntax

```
xs:string codepoints-to-string(xs:integer*)
```

Inputs

A sequence of `xs:integer`s, each of which represents a Unicode codepoint.

Output

An `xs:string` created by concatenating the given codepoints. If the input argument is the empty sequence, `codepoints-to-string( )` returns a zero-length string. If any codepoint in the argument is not a legal XML character, the XSLT processor raises an error.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 7.2, "Functions to Assemble and Disassemble Strings."

Example

Here's a short stylesheet that converts sequences of integers into strings:

```
<?xml version="1.0"?>
<!-- codepoints-to-string.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the codepoints-to-string() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>&#xA;  codepoints-to-string</xsl:text>
    <xsl:text>((76, 105, 108, 121)) = </xsl:text>
    <xsl:value-of
      select="codepoints-to-string((76, 105, 108, 121))"/>

    <xsl:text>&#xA;  codepoints-to-string</xsl:text>
    <xsl:text>((83, 116, 114, 97, 223, 101)) = </xsl:text>
    <xsl:value-of
      select="codepoints-to-string((83, 116, 114, 97, 223, 101))"/>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet generates these results:

```
Tests of the codepoints-to-string() function:

  codepoints-to-string((76, 105, 108, 121)) = Lily
  codepoints-to-string((83, 116, 114, 97, 223, 101)) = Straße
```

We mentioned earlier that if any codepoint in the input sequence is not a valid XML character, the XSLT processor throws an error. As an example, `codepoints-to-string((0))` causes the XSLT processor to crash; feel free to try this in your spare time.

[2.0] collection()

Returns a collection of nodes. This expands on the capabilities of the `document( )` function defined in XSLT 1.0.

Syntax

```
node()* collection(xs:string?)
```

Input

An optional `xs:string` specifying the URI of a collection. If no string is given, the default collection is returned.

Output

The specified collection of nodes. If no collection name is given, the default collection is returned. The details of the URI format and what kinds of resources can be accessed are implementation-dependent.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.5, "Functions and Operators that Generate Sequences."

Example

For our example, we'll use Saxon's implementation that allows us to load an XML document referencing other XML documents. The structure of the collection document is:

```
<?xml version="1.0"?>
<!-- polist.xml -->
<collection>
  <doc href="po38292.xml"/>
  <doc href="po38293.xml"/>
  <doc href="po38294.xml"/>
  <doc href="po38295.xml"/>
</collection>
```

When loading this document with the `collection( )` function, Saxon looks at the `href` attributes of all the `<doc>` elements and considers each one a document URI. Saxon then opens and parses those documents and adds their nodes to the collection. Each document URI in our example points to a purchase order document structured like this:

```
<?xml version="1.0" ?>
<!-- po38293.xml -->
<purchase-order id="38293">
  <date year="2001" month="9" day="8"/>
  <customer id="4738" level="Basic">
    <address type="business">
      <name>
        <title>Ms.</title>
        <first-name>Amanda</first-name>
        <last-name>Reckonwith</last-name>
      </name>
      <street>930-A Chestnut Street</street>
      <city>Lynn</city>
      <state>MA</state>
      <zip>02930</zip>
    </address>
    <address type="ship-to"/>
  </customer>
  <items>
    <item part-no="23813-03-CDK">
      <name>Cucumber Decorating Kit</name>
      <qty>1</qty>
      <price>29.95</price>
    </item>
  </items>
</purchase-order>
```

When the file *polist.xml* is processed, our collection contains four document elements, each of which represents a purchase order. We can then use XSLT and XPath to work with all the nodes in the collection. Here is the stylesheet that loads the collection and works with it:

```
<?xml version="1.0"?>
<!-- collection.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the collection() function:</xsl:text>

    <xsl:variable name="docPile" as="node()*"
      select="collection('polist.xml')"/>

    <xsl:text>&#xA;&#xA;  The customers in the </xsl:text>
    <xsl:text>collection are: &#xA;    </xsl:text>
    <xsl:for-each select="$docPile/purchase-order/customer">
      <xsl:sort select="address/name/last-name"/>
      <xsl:value-of
        select="address/name/title,
                address/name/first-name,
                address/name/last-name"
        separator=" "/>
      <xsl:text>&#xA;    </xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Running the stylesheet gives us these results:

```
A test of the collection() function:

  The customers in the collection are:
    Ms. Natalie Attired
    Mrs. Mary Backstayge
    Mr. Chester Hasbrouck Frisby
    Ms. Amanda Reckonwith
```

We loaded the collection and stored the document nodes of each of the files in the variable `docPile`. Once the collection is loaded, we can work with the nodes in the collection, even though they don't share a common document root. In the sample stylesheet, we listed all of the customers in all of the purchase orders in the collection. We used `<xsl:for-each>` to iterate through the purchase orders, and we used `<xsl:sort>` to list the customers by their last names.

Remember that most of the details of the `collection( )` function are implementation-defined. Saxon's use of the `<collection>` and `<doc>` elements are not part of the spec; neither is the feature that let us load the collection document by typing the name of the file. See the documentation for your XSLT processor for the details of how it implements the `collection( )` function.

[2.0] compare()

Compares two `xs:string`s and returns `-1`, `0`, or `1`, depending on whether the first string is less than, equal to, or greater than the second.

Syntax

```
xs:integer? compare(xs:string?, xs:string?)
xs:integer? compare(xs:string?, xs:string?, $collation as xs:string)
```

Inputs

Two `xs:string`s and an optional collation.

Output

The numeric value `-1`, `0`, or `1`, depending on whether the first string is less than, equal to, or greater than the second. The optional `$collation` argument names a collation used for comparing the two strings. To quote an example from the XSLT 2.0 spec, in a collation for German characters, *Strasse* and *Straße* are equal; in other collations they are not. If either string is the empty sequence, `compare( )` returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 7.3, "Equality and Comparison of Strings."

Example

Here's a stylesheet that uses `compare( )` in several different ways:

```
<?xml version="1.0"?>
<!-- compare1.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Here are some tests of the compare() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>&#xA;  compare('Lily', 'lily') = </xsl:text>
    <xsl:value-of select="compare('Lily', 'lily')"/>

    <xsl:text>&#xA;  compare('Lily', 'Lily') = </xsl:text>
    <xsl:value-of select="compare('Lily', 'Lily')"/>

    <xsl:text>&#xA;  compare('Lily', </xsl:text>
    <xsl:text>'&amp;#x4C;&amp;#x69;&amp;#x6C;&amp;#x79;') = </xsl:text>
    <xsl:value-of select="compare('Lily', '&#x4C;&#x69;&#x6C;&#x79;')"/>

    <xsl:text>&#xA;  compare('Lily', 'Doug') = </xsl:text>
    <xsl:value-of select="compare('Lily', 'Doug')"/>
```

```
    <xsl:text>&#xA;&#xA;  if (not(compare('Lily', 'Lily'))) : </xsl:text>
    <xsl:value-of select="if (not(compare('Lily', 'Lily')))
                          then 'The two test strings are equal!'
                          else 'The two test strings aren''t equal!'"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Here are some tests of the compare() function:

  compare('Lily', 'lily') = -1
  compare('Lily', 'Lily') = 0
  compare('Lily', '&#x4C;&#x69;&#x6C;&#x79;') = 0
  compare('Lily', 'Doug') = 1

  if (not(compare('Lily', 'Lily'))) : The two test strings are equal!
```

The first four tests of the `compare( )` function compared strings, with the third test using character entities instead of the actual text. In the fifth test, we used `compare( )` in a boolean expression; if the two strings are equal, `compare( )` returns `0`. The expression `not(compare( ))` is an awkward way of saying, "If these two strings are not unequal."

Keep in mind that we could use any of the comparison operators in XPath, either the operators for atomic values (`eq`, `ne`, `gt`, `lt`, `ge`, and `le`) or the more general operators (`=`, `!=`, `>`, `<`, `>=`, and `<=`). The only time you're required to use the `compare( )` function is when you need to specify a collation.

Speaking of which, we'll look at a final example that uses a collation function. The German word for *street* can be spelled two ways: *Strasse* and *Straße*. We'll compare the two words twice, once using the default collation and once using a German collation provided by an extension function. Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- compare2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="string1" select="'Stra&#xDF;e'"/>
    <xsl:variable name="string2" select="'Strasse'"/>

    <xsl:text>&#xA;Here are more tests of the compare() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>  compare('</xsl:text>
    <xsl:value-of select="$string1"/>
    <xsl:text>', '</xsl:text>
    <xsl:value-of select="$string2"/>
    <xsl:text>') = </xsl:text>
    <xsl:value-of select="compare($string1, $string2)"/>
    <xsl:text>&#xA;</xsl:text>
```

```
    <xsl:text>  compare('</xsl:text>
    <xsl:value-of select="$string1"/>
    <xsl:text>', '</xsl:text>
    <xsl:value-of select="$string2"/>
    <xsl:text>', [German collation]) = </xsl:text>
    <xsl:value-of
      select="compare($string1, $string2,
              concat('http://saxon.sf.net/collation?',
                     'class=com.oreilly.xslt.GermanCollation;'))"/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Here are more tests of the compare() function:
  compare('Straße', 'Strasse') = 1
  compare('Straße', 'Strasse', [German collation]) = 0
```

As you can see, the two spellings are different in the default collation, but they're equal when we use the German collation. (To keep the listing within the margins of the page, we used the `concat( )` function to combine the two halves of the Saxon collation URI.) See "The document() Function and Sorting" in Chapter 8 for more information.

concat()

Takes all of its arguments and concatenates them. Any arguments that are not strings are converted to strings as if processed by the `string( )` function.

Syntax

[1.0] `string` **`concat(`***`string, string, string*`***`)`**
[2.0] `xs:string` **`concat(`***`xs:anyAtomicType?, xs:anyAtomicType?, ...`***`)`**

Inputs

Two or more strings. The `concat( )` function is unique in that it may have any number of arguments.

[2.0] In XSLT 2.0, the arguments passed to the `concat( )` function can be any atomic type. All of the values are converted to strings. If any of the arguments is the empty sequence, it is treated as a zero-length string. It is an error if any argument is a sequence with a length greater than 1.

Output

The concatenation of all of the input values.

Defined in

[1.0] XPath section 4.2, "String Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 7.4, "Functions on String Values."

Example

We'll use this XML file to demonstrate how `concat( )` works:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

In our stylesheet, we'll use the `concat( )` function to create filenames for various HTML files. The filenames are composed from three pieces of information, concatenated by the `concat( )` function:

```
<?xml version="1.0"?>
<!-- concat.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the concat() function: &#xA;</xsl:text>
    <xsl:for-each select="list/listitem">
      <xsl:text>&#xA;  See the file </xsl:text>
      <xsl:value-of select="concat('album', position(), '.html')"/>
      <xsl:text> to see the details of &#xA;    the album "</xsl:text>
      <xsl:value-of select="."/>
      <xsl:text>."&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Our stylesheet generates these results:

```
Tests of the concat() function:

  See the file album1.html to see the details of
    the album "The Sacred Art of Dub."

  See the file album2.html to see the details of
    the album "Only the Poor Man Feel It."

  See the file album3.html to see the details of
    the album "Excitable Boy."
  ...
```

[2.0] To illustrate how `concat( )` combines atomic values in XSLT 2.0, here's another stylesheet:

```
<?xml version="1.0"?>
<!-- concat2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Another test of the concat() function: &#xA;</xsl:text>

    <xsl:text>&#xA;  concat(3, 4, current-time(), 'blue', </xsl:text>
    <xsl:text>/list/title) = &#xA;    </xsl:text>
    <xsl:value-of
      select="concat(3, 4, current-time(), 'blue', /list/title)"/>
  </xsl:template>

</xsl:stylesheet>
```

When used with our XML input document, the stylesheet produces these results:

```
Another test of the concat() function:

  concat(3, 4, current-time(), 'blue', /list/title) =
    3416:23:12.217-04:00blueAlbums I've bought recently:
```

This concatenates the numbers `3` and `4`, the `xs:time` value returned by the `current-time( )` function, the string `blue`, and the `<title>` element from the input document. All of those values are converted to strings and concatenated.

All the arguments to `concat( )` must be atomic values or sequences of zero or one items. The stylesheet works with our XML input document because there is only one node in the document that matches the XPath expression `/list/title`. If we change the XPath expression to `/list/listitem`, it selects more than one node, and the XSLT processor raises an error. To concatenate sequences of multiple items, use the [2.0] `string-join( )` function.

contains()

Determines whether the first argument string contains the second.

Syntax

```
[1.0] boolean contains(string, string)
[2.0] xs:boolean contains(xs:string?, xs:string?)
[2.0] xs:boolean contains(xs:string?, xs:string?, $collation as xs:string)
```

Inputs

Two strings. If the first string contains the second string, the function returns the boolean value `true`. *[2.0]* In XSLT 2.0, there is an optional third argument—the name of a collation that specifies how strings in different languages are compared.

Output

The boolean value `true` if the first argument contains the second; `false` otherwise. If the second string is a zero-length string, `contains( )` returns `true`. If the first string is a zero-length string, `contains( )` returns `false`.

Defined in

[1.0] XPath section 4.2, "String Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 7.5, "Functions Based on Substring Matching."

Example

This stylesheet uses the `replace-substring` named template. It passes three arguments to the `replace-substring` template: the original string, the substring to be searched for in the original string, and the substring to replace the target substring in the original string. The `replace-substring` template uses the `contains( )`, `substring-after( )`, and `substring-before( )` functions.

Here is our sample stylesheet. It replaces all occurrences of `World` with the string `"Mundo"`:

```
<?xml version="1.0"?>
<!-- contains1.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the contains() function:&#xA;&#xA;</xsl:text>
    <xsl:text>  Replacing 'World' with 'Mundo' in </xsl:text>
    <xsl:text>'Hello World!': &#xA;    </xsl:text>
    <xsl:variable name="test">
      <xsl:call-template name="replace-substring">
        <xsl:with-param name="original">Hello World!</xsl:with-param>
        <xsl:with-param name="substring">World</xsl:with-param>
        <xsl:with-param name="replacement">Mundo</xsl:with-param>
      </xsl:call-template>
    </xsl:variable>
    <xsl:value-of select="$test"/>
  </xsl:template>

  <xsl:template name="replace-substring">
    <xsl:param name="original" />
    <xsl:param name="substring" />
    <xsl:param name="replacement" />
    <xsl:choose>
      <xsl:when test="contains($original, $substring)">
        <xsl:value-of select="substring-before($original, $substring)" />
        <xsl:value-of select="$replacement" />
        <xsl:call-template name="replace-substring">
          <xsl:with-param name="original"
            select="substring-after($original, $substring)" />
```

```
          <xsl:with-param name="substring" select="$substring" />
          <xsl:with-param name="replacement" select="$replacement" />
        </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="$original" />
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

</xsl:stylesheet>
```

In this example, we're replacing one substring with another. To do this, we use three functions (`contains( )`, `substring-after( )`, and `substring-before( )`) and a recursive named template (`replace-substring`). Our recursive template takes three strings as input: the original string, the string to be replaced, and the replacement string itself. If the original string doesn't contain the string to be replaced, our named template returns the entire string. Otherwise, the template calls itself recursively, using three arguments: the portion of the string after the string to be replaced, the string to be replaced, and the replacement string. Put another way, if the named template finds the string to be replaced, it invokes itself on the rest of the string. The `substring-after` function is a key part of the named template.

The stylesheet produces these results, regardless of the XML document used as input:

```
Hello Mundo!
```

[2.0] In our first example, we used XSLT 1.0, so we created a tail-recursive named procedure that uses `contains( )`, `substring-after( )`, and `substring-before( )` to replace one substring with another. In XSLT 2.0, we can use the very useful `replace( )` function instead:

```
<?xml version="1.0"?>
<!-- replace2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the replace() function:&#xA;&#xA;</xsl:text>
    <xsl:text>  Replacing 'World' with 'Mundo' in </xsl:text>
    <xsl:text>'Hello World!': &#xA;    </xsl:text>
    <xsl:value-of select="replace('Hello World', 'World', 'Mundo')"/>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates the same results as our XSLT 1.0 stylesheet, but with far less code.

As a final example, we'll illustrate the use of a collation function:

```
<?xml version="1.0"?>
<!-- contains2.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>
```

```
  <xsl:template match="/">
    <xsl:variable name="string1"
      select="'Sch&#xF6;naicherstra&#xDF;e'"/>
    <xsl:variable name="string2" select="'strasse'"/>

    <xsl:text>&#xA;Another test of the contains() </xsl:text>
    <xsl:text>function:&#xA;&#xA;</xsl:text>

    <xsl:text>  contains('</xsl:text>
    <xsl:value-of select="$string1"/>
    <xsl:text>', '</xsl:text>
    <xsl:value-of select="$string2"/>
    <xsl:text>') = </xsl:text>
    <xsl:value-of select="contains($string1, $string2)"/>
    <xsl:text>&#xA;</xsl:text>

    <xsl:text>  contains('</xsl:text>
    <xsl:value-of select="$string1"/>
    <xsl:text>', '</xsl:text>
    <xsl:value-of select="$string2"/>
    <xsl:text>', [German collation]) = </xsl:text>
    <xsl:value-of
      select="contains($string1, $string2,
              concat('http://saxon.sf.net/collation?',
                     'class=com.oreilly.xslt.GermanCollation;'))"/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Another test of the contains() function:

  contains('Schönaicherstraße', 'strasse') = false
  contains('Schönaicherstraße', 'strasse', [German collation]) = true
```

The German word for *street* can be spelled either *Strasse* or *Straße*. In this stylesheet, using the German collation respects this difference; the default collation does not. (To keep the listing within the margins of the page, we used the `concat( )` function to combine the two halves of the Saxon collation URI.) See "The document() Function and Sorting" in Chapter 8 for more information.

count()

Counts the number of nodes in a given *[1.0]* node-set or *[2.0]* sequence.

Syntax

```
[1.0] number count(node-set)
[2.0] xs:integer count(item()*)
```

Inputs

A node-set or sequence.

Output

The number of items in the node-set or sequence. *[2.0]* If the argument is the empty sequence, `count( )` returns `0`.

Defined in

[1.0] XPath section 4.1, "Node Set Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 15.4, "Aggregate Functions."

Examples

Here's the XML document we'll use to illustrate the `count( )` function:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

And our stylesheet that illustrates the `count( )` function:

```
<?xml version="1.0"?>
<!-- count.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the count() function:&#xA;&#xA;</xsl:text>
```

```
    <xsl:text>  Our store sells </xsl:text>
    <xsl:value-of select="count(/report/brand)"/>
    <xsl:text> different brands of chocolate.</xsl:text>
    <xsl:text>&#xA;&#xA;  For the last month, </xsl:text>
    <xsl:value-of select="count(/report/brand[units > 20000])"/>
    <xsl:text> brand(s) sold more than 20,000 units,</xsl:text>
    <xsl:text>&#xA;    and all but </xsl:text>
    <xsl:value-of select="count(/report/brand[units &lt; 10000])"/>
    <xsl:text> brand(s) sold 10,000 units or more.</xsl:text>

    <xsl:text>&#xA;&#xA;  Here are the brands we sell: </xsl:text>
    <xsl:for-each select="/report/brand">
      <xsl:text>&#xA;    Brand </xsl:text>
      <xsl:value-of select="position()"/>
      <xsl:text> of </xsl:text>
      <xsl:value-of select="count(/report/brand)"/>
      <xsl:text>: </xsl:text>
      <xsl:value-of select="name"/>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results of our stylesheet:

```
Tests of the count() function:

  Our store sells 5 different brands of chocolate.

  For the last month, 2 brand(s) sold more than 20,000 units,
    and all but 1 brand(s) sold 10,000 units or more.

  Here are the brands we sell:
    Brand 1 of 5: Lindt
    Brand 2 of 5: Callebaut
    Brand 3 of 5: Valrhona
    Brand 4 of 5: Perugina
    Brand 5 of 5: Ghirardelli
```

We use the `count( )` function in several different ways here. In the first sentence, the number `5` is the count of `<brand>` elements in the document. In the second sentence, we use `count( )` with XPath predicates to count only `<brand>` elements whose `<units>` children have certain values. Finally, in the list of the brands we sell, we use the `position( )` and `count( )` functions together to create the "Brand x of y" text.

There are two things to notice. First of all, the stylesheet begins with `<xsl:stylesheet version="1.0"`. This stylesheet works with both XSLT 1.0 and 2.0 processors. Secondly, we used `<` for the less-than symbol (<); the value of an attribute can't contain a less-than sign.

Here's a short stylesheet to illustrate how `count( )` works with sequences in XSLT 2.0:

```
<?xml version="1.0"?>
<!-- count2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xsl:output method="text"/>

    <xsl:template match="/">
      <xsl:text>&#xA;Tests of the count() function:&#xA;&#xA;</xsl:text>

      <xsl:variable name="salesFigures" as="xs:integer*">
        <xsl:sequence select="/report/brand/units"/>
      </xsl:variable>

      <xsl:text>  Our store sells </xsl:text>
      <xsl:value-of select="count($salesFigures)"/>
      <xsl:text> different brands of chocolate.</xsl:text>
      <xsl:text>&#xA;&#xA;  For the last month, </xsl:text>
      <xsl:value-of select="count($salesFigures[. > 20000])"/>
      <xsl:text> brand(s) sold more than 20,000 units,</xsl:text>
      <xsl:text>&#xA;    and all but </xsl:text>
      <xsl:value-of select="count($salesFigures[. &lt; 10000])"/>
      <xsl:text> brand(s) sold 10,000 units or more.</xsl:text>
    </xsl:template>

</xsl:stylesheet>
```

We create a sequence of `xs:integer`s here, and then use the `count( )` function to count the items in the sequence. Just as we did with the first stylesheet, we use predicates to select certain values from the sequence. Here are the results:

```
Tests of the count() function:

  Our store sells 5 different brands of chocolate.

  For the last month, 2 brand(s) sold more than 20,000 units,
    and all but 1 brand(s) sold 10,000 units or more.
```

In the second stylesheet, our sequence contains the `xs:integer` values of the `<units>` elements in our source document, so the XPath predicate uses the current node (`.`) in the comparison expression: `count($salesFigures[. > 20000]`. In our first stylesheet, we could have written `count(/report/brand[units < 10000])` as `count(/report/brand/units[. < 10000]` to get the same results with a similar XPath expression.

current()

Returns a node-set that has the current node as its only member.

Syntax

[1.0] node-set **current()**
[2.0] item() **current()**

Inputs

None.

Output

[1.0] A node-set that has the current node as its only member. Most of the time, the current node is no different than the context node.

[2.0] This function, used inside an XPath expression, returns the item that was the context item when the expression was invoked.

In XSLT 1.0 and 2.0, the current node/item is almost always the same as the context node/item. For example, the two XPath expressions below are always the same:

```
<xsl:value-of select="current()"/>
<xsl:value-of select="."/>
```

Within a predicate expression, however, the current node and the context node are usually different.

Defined in

[1.0] XSLT section 12.4, "Miscellaneous Additional Functions."

[2.0] XSLT section 16.6, "Miscellaneous Additional Functions."

Example

We'll use the `current( )` function along with a lookup table. We'll transform a modified version of our chocolate sales figures:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate-by-month.xml -->
<report year="2006">
  <title>Chocolate bar sales by month</title>
  <sales month="08" total="91316"/>
  <sales month="09" total="82911"/>
  <sales month="10" total="128587"/>
  <sales month="11" total="79244"/>
  <sales month="12" total="113606"/>
</report>
```

Here's our stylesheet. It has a lookup table of month names; we'll use the lookup table to replace the `month` attribute with the name of the corresponding month. The stylesheet does the same transform three times, once with the `current( )` function and twice without it:

```
<?xml version="1.0"?>
<!-- current.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:months="http://www.months.com">

  <months:name sequence="01">January</months:name>
  <months:name sequence="02">February</months:name>
  <months:name sequence="03">March</months:name>
  <months:name sequence="04">April</months:name>
  <months:name sequence="05">May</months:name>
  <months:name sequence="06">June</months:name>
  <months:name sequence="07">July</months:name>
  <months:name sequence="08">August</months:name>
```

```
  <months:name sequence="09">September</months:name>
  <months:name sequence="10">October</months:name>
  <months:name sequence="11">November</months:name>
  <months:name sequence="12">December</months:name>

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the current() function:&#xA;</xsl:text>

    <xsl:text>&#xA;  Using the current() function:&#xA;</xsl:text>
    <xsl:text>&#xA;    Chocolate bar sales for </xsl:text>
    <xsl:value-of select="/report/@year"/>
    <xsl:text>: &#xA;</xsl:text>

    <xsl:for-each select="/report/sales">
      <xsl:text>&#xA;      </xsl:text>
      <xsl:value-of
        select="document('')/*/months:name[@sequence=current()/@month][1]"/>
      <xsl:text> - Total sales = </xsl:text>
      <xsl:value-of select="format-number(@total, '##,###')"/>
    </xsl:for-each>

    <xsl:text>&#xA;&#xA;  Without the current() function:&#xA;</xsl:text>
    <xsl:text>&#xA;    Chocolate bar sales for </xsl:text>
    <xsl:value-of select="/report/@year"/>
    <xsl:text>: &#xA;</xsl:text>

    <xsl:for-each select="/report/sales">
      <xsl:text>&#xA;      </xsl:text>
      <xsl:value-of
        select="document('')/*/months:name[@sequence=./@month][1]"/>
      <xsl:text> - Total sales = </xsl:text>
      <xsl:value-of select="format-number(@total, '##,###')"/>
    </xsl:for-each>

    <xsl:text>&#xA;&#xA;  Another test without the current() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>
    <xsl:text>&#xA;    Chocolate bar sales for </xsl:text>
    <xsl:value-of select="/report/@year"/>
    <xsl:text>: &#xA;</xsl:text>

    <xsl:for-each select="/report/sales">
      <xsl:text>&#xA;      </xsl:text>
      <xsl:value-of
        select="document('')/*/months:name[@sequence=./@sequence][1]"/>
      <xsl:text> - Total sales = </xsl:text>
      <xsl:value-of select="format-number(@total, '##,###')"/>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Tests of the current() function:

  Using the current() function:

    Chocolate bar sales for 2006:

      August - Total sales = 91,316
      September - Total sales = 82,911
      October - Total sales = 128,587
      November - Total sales = 79,244
      December - Total sales = 113,606

  Without the current() function:

    Chocolate bar sales for 2006:

       - Total sales = 91,316
       - Total sales = 82,911
       - Total sales = 128,587
       - Total sales = 79,244
       - Total sales = 113,606

  Another test without the current() function:

    Chocolate bar sales for 2006:

      January - Total sales = 91,316
      January - Total sales = 82,911
      January - Total sales = 128,587
      January - Total sales = 79,244
      January - Total sales = 113,606
```

The first test works correctly because we use the `current( )` function. This function returns the `<sales>` element we're currently processing. Within the predicate expression, we have to use `current( )` to get that element. The `<xsl:value-of>` element outputs the text of the `<months:name>` element whose `sequence` attribute matches the `month` attribute of the current `<sales>` element.

Notice that we end the XPath expression with another predicate, `[1]`, to make sure we select only the first match. That doesn't matter in XSLT 1.0, because `<xsl:value-of>` uses only the first node in the node-set. XSLT 2.0, however, uses *all* the items that match. Because of the structure of our XML source document, only one item matches in the first test, but that might not be the case for all XML source documents. For that reason, specifying exactly which node you want to use is a good habit to get into, especially if you're going to migrate XSLT 1.0 stylesheets to XSLT 2.0.

The second and third tests don't work because we don't use the `current( )` function. In the second test, our XPath expression is `document('')/*/months:name[@sequence=./@month][1]`. The dot here represents the `<months:name>` element we're evaluating, *not* the current `<sales>` element. The XPath predicate `[@sequence=./@month]` doesn't match anything, because there aren't any `<months:name>` elements that have a `month` attribute.

The point of the third test is to illustrate what the dot operator (the context node) actually represents. The predicate `[@sequence=./@sequence]` wouldn't match anything if the dot represented the `<sales>` element we're currently processing; there aren't any `<sales>` elements with a `sequence` attribute. Because the dot represents the `<months:name>` element, there is a match. We specified that XSLT should output the first item (`[1]`), so every month is January.

A final note about the predicate specifying that we use the first item: in the first test, we could leave out the `[1]` predicate because the XPath expression matches only one node based on our XML source document. In the second test, we could leave out the `[1]` predicate because there weren't any matches. In the third test, however, leaving out the `[1]` predicate and changing the stylesheet version to `2.0` gives these results:

```
  Another test without the current() function:

    Chocolate bar sales for 2006:

      January February March April May June July August September Octo
ber November December - Total sales = 91,316
      January February March April May June July August September Octo
ber November December - Total sales = 82,911
      January February March April May June July August September Octo
ber November December - Total sales = 128,587
      January February March April May June July August September Octo
ber November December - Total sales = 79,244
      January February March April May June July August September Octo
ber November December - Total sales = 113,606
```

In XSLT 2.0, `<xsl:value-of>` uses all of the nodes that match, not just the first.

[2.0] current-date()

Returns the current date as an `xs:date` value.

Syntax

```
xs:date current-date()
```

Inputs

None.

Output

An `xs:date` value set to the current time on the system. The returned value is set to the default timezone. (You can get the details of the default timezone with the `implicit-timezone( )` function). *The* `current-date( )` *function returns the same value throughout the processing of the stylesheet.* If you call `current-date( )` a dozen times throughout your stylesheet, the function returns the same value each time.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 16, "Context Functions."

Example

Here is a stylesheet that displays the current date. It uses the `current-date( )`, `format-date( )`, and `implicit-timezone( )` functions.

```
<?xml version="1.0"?>
<!-- current-date.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>The current date is </xsl:text>
    <xsl:value-of
      select="format-date(current-date(),
              '[FNn], the [D1o] of [MNn], [Y01]')"/>
    <xsl:text>&#xA;&#xA;The implicit timezone for the </xsl:text>
    <xsl:text>current context is: </xsl:text>
    <xsl:value-of select="implicit-timezone()"/>
    <xsl:text>&#xA;&#xA;The timezone extracted from the </xsl:text>
    <xsl:text>current date is: </xsl:text>
    <xsl:value-of select="timezone-from-date(current-date())"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the stylesheet's results:

```
The current date is Thursday, the 6th of July, 06

The implicit timezone for the current context is: -PT4H

The timezone extracted from the current date is: -PT4H
```

Notice that the implicit timezone retrieved with the `implicit-timezone( )` function is the same timezone we extracted from the current date. The *implicit timezone* is a component of all the `xs:date`, `xs:dateTime`, and `xs:time` values created without specifying a particular timezone.

[2.0] current-dateTime()

Returns the current date and time as an `xs:dateTime` value.

Syntax

```
xs:dateTime current-dateTime()
```

Inputs

None.

Output

An `xs:dateTime` value set to the current date and time on the system. The returned value is set to the default timezone. (You can get the details of the default timezone with the context

function `implicit-timezone( )`). *The `current-dateTime( )` function returns the same value throughout the processing of the stylesheet.* If you call `current-dateTime( )` a dozen times throughout your stylesheet, the function returns the same value each time.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 16, "Context Functions."

Example

Here is a stylesheet that displays the current date and time. It uses the `current-dateTime( )`, `format-dateTime( )` and `implicit-timezone( )` functions.

```
<?xml version="1.0"?>
<!-- current-datetime.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>The current date and time is </xsl:text>
    <xsl:value-of
      select="format-dateTime(current-dateTime(),
              '[h]:[m01] [Pn] on [FNn], the [D1o] of [MNn], [Y]')"/>
    <xsl:text>&#xA;&#xA;The implicit timezone for the </xsl:text>
    <xsl:text>current context is: </xsl:text>
    <xsl:value-of select="implicit-timezone()"/>
    <xsl:text>&#xA;&#xA;The timezone extracted from the </xsl:text>
    <xsl:text>current date and time is: </xsl:text>
    <xsl:value-of select="timezone-from-dateTime(current-dateTime())"/>
  </xsl:template>

</xsl:stylesheet>
```

The results of this stylesheet are:

```
The current date and time is 6:43 a.m. on Thursday, the 6th of July, 2006

The implicit timezone for the current context is: -PT4H

The timezone extracted from the current date and time is: -PT4H
```

Notice that the implicit timezone retrieved with the `implicit-timezone( )` function is the same timezone we extracted from the current date and time. The implicit timezone is a component of all the `xs:date`, `xs:dateTime`, and `xs:time` values created without specifying a particular timezone.

[2.0] current-group()

Returns all the items that are included in the current group.

Syntax

```
item()* current-group()
```

Inputs

None.

Output

The set of items that are included in the current group. This function is only useful inside the `<xsl:for-each-group>` element. Calling `current-group( )` outside `<xsl:for-each-group>` returns the empty sequence.

Defined in

XSLT 2.0 section 14, "Grouping."

Example

We'll use a simplified version of our address book document to illustrate the `current-group( )` function:

```
<?xml version="1.0"?>
<!-- simple-addresses.xml -->
<addressbook>
  <address>
    <name>Mr. Chester Hasbrouck Frisby</name>
    <city>Sheboygan</city>
    <state>WI</state>
  </address>
  <address>
    <name>Mary Backstayge</name>
    <city>Skunk Haven</city>
    <state>MA</state>
  </address>
  <address>
    <name>Ms. Natalie Attired</name>
    <city>Winter Harbor</city>
    <state>ME</state>
  </address>
  <address>
    <name>Harry Backstayge</name>
    <city>Skunk Haven</city>
    <state>MA</state>
  </address>
  <address>
    <name>Mary McGoon</name>
    <city>Boylston</city>
    <state>VA</state>
  </address>
  <address>
    <name>Ms. Amanda Reckonwith</name>
    <city>Lynn</city>
    <state>MA</state>
```

```
  </address>
</addressbook>
```

We'll group these names by `<state>` with this stylesheet:

```
<?xml version="1.0"?>
<!-- current-group.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Here's a test of the </xsl:text>
    <xsl:text>current-group() function:&#xA;</xsl:text>
    <xsl:text>&#xA;  Customers grouped by state</xsl:text>

    <xsl:for-each-group select="//address" group-by="state">
      <xsl:sort select="state"/>
      <xsl:text>&#xA;&#xA;    Customers in </xsl:text>
      <xsl:value-of select="current-grouping-key()"/>
      <xsl:text>:</xsl:text>

      <xsl:for-each select="current-group()">
        <xsl:text>&#xA;      </xsl:text>
        <xsl:value-of select="name"/>
        <xsl:text> of </xsl:text>
        <xsl:value-of select="city"/>
      </xsl:for-each>
    </xsl:for-each-group>
  </xsl:template>

</xsl:stylesheet>
```

In this example, we're grouping the people in our address book by the state they live in. The groups are selected by the `<xsl:for-each-group>` element. Within this element, we use `current-group( )` in an `<xsl:for-each>` element to iterate through all the items in the current group.

Running our stylesheet against our simplified address book generates these results:

```
Here's a test of the current-group() function:

  Customers grouped by state:

    Customers in MA:
      Mary Backstayge of Skunk Haven
      Harry Backstayge of Skunk Haven
      Ms. Amanda Reckonwith of Lynn

    Customers in ME:
      Ms. Natalie Attired of Winter Harbor

    Customers in VA:
      Mary McGoon of Boylston
```

```
Customers in WI:
  Mr. Chester Hasbrouck Frisby of Sheboygan
```

We could make our stylesheet more sophisticated by using the `count( )` function with `current-group( )`. If we have more than one customer in a state (`count(current-group( )) > 1`), our heading could be `Customers in MA`. For states that have only one customer (`count(current-group( )) = 1`), our heading could be `Our only customer in ME`.

[2.0] current-grouping-key()

Returns the value used to select the current group.

Syntax

```
xs:anyAtomicType? current-grouping-key()
```

Inputs

None.

Output

The value used to select the items in the current group. The `current-grouping-key( )` function is only useful inside an `<xsl:for-each-group>` element with a `group-by` or `group-adjacent` attribute. Calling `current-grouping-key( )` anywhere else returns the empty sequence.

Defined in

XSLT 2.0 section 14, "Grouping."

Example

We'll use a simplified version of our address book document to illustrate the `current-grouping-key( )` function:

```
<?xml version="1.0"?>
<!-- simple-addresses.xml -->
<addressbook>
  <address>
    <name>Mr. Chester Hasbrouck Frisby</name>
    <city>Sheboygan</city>
    <state>WI</state>
  </address>
  <address>
    <name>Mary Backstayge</name>
    <city>Skunk Haven</city>
    <state>MA</state>
  </address>
  <address>
    <name>Ms. Natalie Attired</name>
    <city>Winter Harbor</city>
    <state>ME</state>
  </address>
  <address>
```

```
    <name>Harry Backstayge</name>
    <city>Skunk Haven</city>
    <state>MA</state>
  </address>
  <address>
    <name>Mary McGoon</name>
    <city>Boylston</city>
    <state>VA</state>
  </address>
  <address>
    <name>Ms. Amanda Reckonwith</name>
    <city>Lynn</city>
    <state>MA</state>
  </address>
</addressbook>
```

We'll group these names by `<state>` with this stylesheet:

```
<?xml version="1.0"?>
<!-- current-grouping-key.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Here's a test of the </xsl:text>
    <xsl:text>current-grouping-key() function:&#xA;</xsl:text>
    <xsl:text>&#xA;  Customers grouped by state</xsl:text>

    <xsl:for-each-group select="//address" group-by="state">
      <xsl:sort select="state"/>
      <xsl:text>&#xA;&#xA;    Customers in </xsl:text>
      <xsl:value-of select="current-grouping-key()"/>
      <xsl:text>:</xsl:text>

      <xsl:for-each select="current-group()">
        <xsl:text>&#xA;      </xsl:text>
        <xsl:value-of select="name"/>
        <xsl:text> of </xsl:text>
        <xsl:value-of select="city"/>
      </xsl:for-each>
    </xsl:for-each-group>
  </xsl:template>

</xsl:stylesheet>
```

In this example, we're grouping the people in our address book by the state they live in. In the heading for each group, we output the grouping value (the state) with `current-grouping-key( )` in the heading for each state:

```
Here's a test of the current-grouping-key() function:

  Customers grouped by state:

    Customers in MA:
```

```
    Mary Backstayge of Skunk Haven
    Harry Backstayge of Skunk Haven
    Ms. Amanda Reckonwith of Lynn

  Customers in ME:
    Ms. Natalie Attired of Winter Harbor

  Customers in VA:
    Mary McGoon of Boylston

  Customers in WI:
    Mr. Chester Hasbrouck Frisby of Sheboygan
```

[2.0] current-time()

Returns an `xs:time` object set to the current time.

Syntax

```
xs:time current-time()
```

Inputs

None.

Output

An `xs:time` value set to the current time on the system. The returned value is set to the default timezone. (You can get the details of the default timezone with the context function `implicit-timezone( )`). *The* `current-time( )` *function returns the same value throughout the processing of the stylesheet.* If you call `current-time( )` a dozen times throughout your stylesheet, the function returns the same value each time.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 2, "Accessors."

Example

Here is a stylesheet that displays the current time. It uses the `current-time( )`, `format-time( )`, and `implicit-timezone( )` functions:

```
<?xml version="1.0"?>
<!-- current-time.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>The current time is </xsl:text>
    <xsl:value-of
      select="format-time(current-time(),
              '[h]:[m01] [Pn]')"/>
```

```
    <xsl:text>&#xA;&#xA;The implicit timezone for the </xsl:text>
    <xsl:text>current context is: </xsl:text>
    <xsl:value-of select="implicit-timezone()"/>
    <xsl:text>&#xA;&#xA;The timezone extracted from the </xsl:text>
    <xsl:text>current date and time is: </xsl:text>
    <xsl:value-of select="timezone-from-time(current-time())"/>
  </xsl:template>

</xsl:stylesheet>
```

The exciting results from this stylesheet look like this:

```
The current time is 6:49 a.m.

The implicit timezone for the current context is: -PT4H

The timezone extracted from the current date and time is: -PT4H
```

Notice that the implicit timezone retrieved with the `implicit-timezone( )` function is the same timezone we extracted from the current time. The implicit timezone is a component of all the `xs:date`, `xs:dateTime`, and `xs:time` values created without specifying a particular timezone.

[2.0] data()

Given a sequence of items, returns a sequence of atomic values that represent the set of items.

Syntax

```
xs:anyAtomicType* data(item()*)
```

Inputs

A sequence of items.

Output

A sequence in which each item in the input sequence has been converted to an atomic value. If an item in the input sequence is an atomic value, it is returned as is in the result sequence. For nodes, any item with a datatype is converted to an atomic value and returned in the result sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 2, "Accessors."

Example

Here is a stylesheet that creates a sequence of atoms and elements and lists them, identifying atomic values versus elements in the sequence. Next, the stylesheet lists the sequence returned by the `data( )` function:

```
<?xml version="1.0"?>
<!-- data.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:datatest="http://www.oreilly.com">

<xsl:output method="text"/>

<xsl:template match="/">

  <!-- Before we test the data() function, we create a sequence. -->
  <xsl:variable name="testSequence" as="item()*">
    <xsl:sequence select="(3, 4, 5)"/>
    <xsl:element name="currentDate">
      <xsl:value-of select="current-date()"/>
    </xsl:element>
    <xsl:element name="currentTime">
      <xsl:value-of select="current-time()"/>
    </xsl:element>
    <xsl:element name="integerTest">
      <xsl:value-of select="xs:integer(8)"/>
    </xsl:element>
    <xsl:sequence select="('blue', 'red')"/>
    <xsl:element name="floatTest1">
      <xsl:value-of select="xs:float(3.14)"/>
    </xsl:element>
    <xsl:element name="floatTest2">
      <xsl:value-of select="xs:float(42)"/>
    </xsl:element>
    <xsl:element name="dateTest">
      <xsl:value-of select="xs:date('1995-04-21')"/>
    </xsl:element>
  </xsl:variable>

  <xsl:text>&#xA;Here is a test of the data() </xsl:text>
  <xsl:text>function:&#xA;</xsl:text>

  <xsl:text>&#xA;  Our original sequence is:&#xA;&#xA;    </xsl:text>
  <xsl:value-of
    select="for $i in (1 to count($testSequence))
            return (datatest:print-item(
            subsequence($testSequence, $i, 1)))"
    separator="&#xA;    "/>

  <xsl:text>&#xA;&#xA;  Passing our sequence to data() </xsl:text>
  <xsl:text>gives us:&#xA;&#xA;    </xsl:text>

  <xsl:variable name="atomicSequence" as="item()*"
    select="data($testSequence)"/>
  <xsl:value-of
    select="for $i in (1 to count($atomicSequence))
            return (datatest:print-item(
            subsequence($atomicSequence, $i, 1)))"
    separator="&#xA;    "/>
</xsl:template>

<!-- Given an item(), this function returns a string -->
<!-- describing that item. -->
```

```
  <xsl:function name="datatest:print-item" as="xs:string">
    <xsl:param name="item" as="item()"/>
    <xsl:choose>
      <xsl:when test="$item instance of element()">
        <xsl:value-of
          select="concat('Element:         &lt;', name($item), '&gt;', $item,
                  '&lt;/', name($item), '&gt;')"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of
          select="concat('Atomic value:    ', $item)"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:function>

</xsl:stylesheet>
```

Here are the results:

```
Here is a test of the data() function:

  Our original sequence is:

    Atomic value:    3
    Atomic value:    4
    Atomic value:    5
    Element:         <currentDate>2008-03-04-05:00</currentDate>
    Element:         <currentTime>23:58:38.609-05:00</currentTime>
    Element:         <integerTest>8</integerTest>
    Atomic value:    blue
    Atomic value:    red
    Element:         <floatTest1>3.14</floatTest1>
    Element:         <floatTest2>42</floatTest2>
    Element:         <dateTest>1995-04-21</dateTest>

  Passing our sequence to data() gives us:

    Atomic value:    3
    Atomic value:    4
    Atomic value:    5
    Atomic value:    2008-03-04-05:00
    Atomic value:    23:58:38.609-05:00
    Atomic value:    8
    Atomic value:    blue
    Atomic value:    red
    Atomic value:    3.14
    Atomic value:    42
    Atomic value:    1995-04-21
```

The test sequence we create has 11 items, 5 of which are atoms and 6 of which are elements. We pass each item in the sequence to our `datatest:print-item( )` function. This function looks at the type of the item. If it is an `element( )`, the function returns a string that includes the opening tag for the element, its value, and its closing tag. For atoms, the function simply returns the atom's value. All of the strings returned by `datatest:print-item( )` are labeled

with `Element:` or `Atom:` so we can see whether each item is an atom or an element. This shows us the contents of the original sequence.

After listing all the atoms and elements in the original sequence, we use the `data( )` function to convert our original sequence to a sequence of atomic values. Calling our `datatest:print-item( )` against each item in the new sequence shows that it is composed of atoms only.

In our stylesheet, we invoke our `datatest:print-item( )` function against each item with the following XPath expression:

```
select="for $i in (1 to count($testSequence))
        return (datatest:print-item(
        subsequence($testSequence, $i, 1)))"
```

This expression uses the new XPath `for` instruction to iterate over the items in the sequence (`count($testSequence)`). For each item, we return the `xs:string` value returned by our `datatest:print-item( )` function. To keep the stylesheet organized, we have the relatively complicated XPath expressions in the `match="/"` template, and we create a function to handle the repetitive task of printing each item.

If we want, we can create an XPath expression that does the work of the function as well. Here's what that code looks like:

```
<!-- data2.xsl -->
...
    <xsl:for-each select="$testSequence">
      <xsl:value-of
        select="if (. instance of node())
                then concat('Element:         &lt;', name(.),
                            '&gt;', ., '&lt;/', name(.), '&gt;')
                else concat('Atomic value:    ', .)"/>
      <xsl:text>&#xA;    </xsl:text>
    </xsl:for-each>
...
    <xsl:variable name="atomicSequence" as="item()*"
      select="data($testSequence)"/>
    <xsl:for-each select="$atomicSequence">
      <xsl:value-of
        select="if (. instance of item())
                then concat('Atomic value:    ', .)
                else concat('Element:         &lt;', name(.),
                            '&gt;', ., '&lt;/', name(.), '&gt;')"/>
      <xsl:text>&#xA;    </xsl:text>
    </xsl:for-each>
...
```

This works, but it requires us to duplicate the `if-then-else` logic wherever we want to print a sequence. It's much better to use an `<xsl:function>` to do the repetitive work. There's another concern: an XSLT processor can give warnings about the type safety of the calls to the `name( )` function. If we change the processing of `$atomicSequence` so that it begins `. instance of node( )`, we get a static error and the stylesheet won't work. To avoid this, it's simpler to put the code into a function.

[2.0] dateTime()

This is a special constructor function that lets you create an `xs:dateTime` value from an `xs:date` and `xs:time`.

Syntax

```
xs:dateTime? dateTime(xs:date?, xs:time?)
```

Inputs

An `xs:date` value and an `xs:time` value.

Output

A new `xs:dateTime` value based on the two input values. If either input value is the empty sequence, `xs:dateTime( )` returns the empty sequence. The timezone of the `xs:dateTime` result is calculated as follows:

- If neither argument has a timezone, the result does not have a timezone.
- If one of the arguments has a timezone or if both arguments have the same timezone, the result has that timezone.
- If the two arguments have different timezones, the XSLT processor raises an error.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 5, "Constructor Functions."

Example

This stylesheet uses the values from the `current-date( )` and `current-time( )` functions to create a new `xs:dateTime` value:

```
<?xml version="1.0"?>
<!-- datetime.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Creating an xs:dateTime with an </xsl:text>
    <xsl:text>xs:date and xs:time:</xsl:text>
    <xsl:variable name="currentDate" as="xs:date" select="current-date()"/>
    <xsl:variable name="currentTime" as="xs:time" select="current-time()"/>
    <xsl:text>&#xA;&#xA;  The current date is: </xsl:text>
    <xsl:value-of select="$currentDate"/>
    <xsl:text>&#xA;&#xA;  The current time is: </xsl:text>
    <xsl:value-of select="$currentTime"/>

    <xsl:variable name="currentDateTime"
      select="dateTime($currentDate, $currentTime)"/>
```

```
    <xsl:text>&#xA;&#xA;  The new xs:dateTime is: </xsl:text>
    <xsl:value-of select="$currentDateTime"/>
    <xsl:text>&#xA;&#xA;  The new xs:dateTime value can be </xsl:text>
    <xsl:text>written as &#xA;    </xsl:text>
    <xsl:value-of
      select="format-dateTime($currentDateTime,
              '[h]:[m01] [Pn] on [FNn], the [D1o] of [MNn], [Y0001]')"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Creating an xs:dateTime with an xs:date and xs:time:

  The current date is: 2006-11-16-05:00

  The current time is: 03:48:09.888-05:00

  The new xs:dateTime is: 2006-11-16T03:48:09.888-05:00

  The new xs:dateTime value can be written as
    3:48 a.m. on Thursday, the 16th of November, 2006
```

[2.0] day-from-date()

Given an `xs:date` value, returns its day value.

Syntax

```
xs:integer? day-from-date(xs:date?)
```

Input

An `xs:date` value.

Output

An `xs:integer` representing the day component of the given `xs:date` value. If the input argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `day-from-date( )` function:

```
<?xml version="1.0"?>
<!-- day-from-date.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>
```

```
  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the day from an xs:date:</xsl:text>
    <xsl:variable name="currentDate" as="xs:date" select="current-date()"/>
    <xsl:text>&#xA;&#xA;The current date is: </xsl:text>
    <xsl:value-of select="$currentDate"/>

    <xsl:text>&#xA;&#xA;  The current day: </xsl:text>
    <xsl:value-of select="day-from-date($currentDate)"/>
    <xsl:text>&#xA;    In words: </xsl:text>
    <xsl:value-of select="format-date($currentDate, '[DWw]')"/>
    <xsl:text>&#xA;    In German: </xsl:text>
    <xsl:value-of select="format-date($currentDate, '[Dw]', 'de', (), ())"/>
    <xsl:text>&#xA;    It's the </xsl:text>
    <xsl:value-of select="format-date($currentDate, '[dwo]')"/>
    <xsl:text> day of the year.</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet creates these results:

```
Extracting the day from an xs:date:

The current date is: 2006-11-16-05:00

  The current day: 16
    In words: Sixteen
    In German: sechszehn
    It's the three hundred and twentieth day of the year.
```

Notice that some of the results were generated by the `day-from-date( )` extraction function, while other results were generated by using `format-date( )` with a format string that selected only the day component of the `xs:date` value.

See Also

The definitions of the [2.0] `format-date( )`, [2.0] `month-from-date( )`, [2.0] `timezone-from-date( )`, and [2.0] `year-from-date( )` functions.

[2.0] day-from-dateTime()

Given an `xs:dateTime` value, returns its day value.

Syntax

```
xs:integer? day-from-dateTime(xs:dateTime?)
```

Inputs

An `xs:dateTime` value.

Output

An `xs:integer` representing the day component of the given `xs:dateTime` value. If the input argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `day-from-dateTime( )` function:

```
<?xml version="1.0"?>
<!-- day-from-datetime.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the day from an xs:dateTime:</xsl:text>
    <xsl:variable name="currentDateTime" as="xs:dateTime"
      select="current-dateTime()"/>
    <xsl:text>&#xA;&#xA;The current date and time is: </xsl:text>
    <xsl:value-of select="$currentDateTime"/>

    <xsl:text>&#xA;&#xA;  The current day: </xsl:text>
    <xsl:value-of select="day-from-dateTime($currentDateTime)"/>
    <xsl:text>&#xA;    In words: </xsl:text>
    <xsl:value-of select="format-dateTime($currentDateTime, '[DWw]')"/>
    <xsl:text>&#xA;    In German words: </xsl:text>
    <xsl:value-of
      select="format-dateTime($currentDateTime, '[Dw]', 'de', (), ())"/>
    <xsl:text>&#xA;    It's the </xsl:text>
    <xsl:value-of select="format-dateTime($currentDateTime, '[dwo]')"/>
    <xsl:text> day of the year.</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet creates these results:

```
Extracting the day from an xs:dateTime:

The current date and time is: 2006-11-16T04:04:46.811-05:00

  The current day: 16
    In words: Sixteen
    In German words: sechszehn
    It's the three hundred and twentieth day of the year.
```

Notice that some of the results were generated by the `day-from-dateTime( )` function, while others were generated by using `format-dateTime( )` with a format string that selected only the day component of the `xs:dateTime` value.

See Also

The definitions of the [2.0] `format-dateTime( )`, [2.0] `hours-from-dateTime( )`, [2.0] `minutes-from-dateTime( )`, [2.0] `month-from-dateTime( )`, [2.0] `seconds-from-dateTime( )`, [2.0] `time zone-from-dateTime( )`, and [2.0] `year-from-dateTime( )` functions.

[2.0] days-from-duration()

Given an `xs:duration` value, returns the number of days in that duration.

Syntax

```
xs:integer? days-from-duration(xs:duration?)
```

Inputs

A `xs:duration`.

Output

An `xs:integer` representing the days component of the given `xs:duration`. Be aware that for an `xs:yearMonthDuration`, this function always returns `0` because there is no days component of an `xs:yearMonthDuration`. Also, if the input argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `days-from-duration( )` function with all three types of durations:

```
<?xml version="1.0"?>
<!-- days-from-duration.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the days component from durations:</xsl:text>

    <xsl:variable name="sampleDuration" as="xs:duration"
      select="xs:duration('P3Y8M2DT4H23M12.2S')"/>
    <xsl:variable name="sampleYearMonthDuration" as="xs:yearMonthDuration"
```

```
      select="xs:yearMonthDuration('P3Y8M')"/>
    <xsl:variable name="sampleDayTimeDuration" as="xs:dayTimeDuration"
      select="xs:dayTimeDuration('P2DT4H23M12.2S')"/>

    <xsl:text>&#xA;&#xA;  A sample xs:duration: </xsl:text>
    <xsl:value-of select="$sampleDuration"/>
    <xsl:text>&#xA;    The days component of this duration is </xsl:text>
    <xsl:value-of select="days-from-duration($sampleDuration)"/>
    <xsl:text>.</xsl:text>

    <xsl:text>&#xA;&#xA;  A sample xs:yearMonthDuration: </xsl:text>
    <xsl:value-of select="$sampleYearMonthDuration"/>
    <xsl:text>&#xA;    The days component of this duration is </xsl:text>
    <xsl:value-of select="days-from-duration($sampleYearMonthDuration)"/>
    <xsl:text>.</xsl:text>

    <xsl:text>&#xA;&#xA;  A sample xs:dayTimeDuration: </xsl:text>
    <xsl:value-of select="$sampleDayTimeDuration"/>
    <xsl:text>&#xA;    The days component of this duration is </xsl:text>
    <xsl:value-of select="days-from-duration($sampleDayTimeDuration)"/>
    <xsl:text>.</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates these results:

```
Extracting the days component from durations:

  A sample xs:duration: P3Y8M2DT4H23M12.2S
    The days component of this duration is 2.

  A sample xs:yearMonthDuration: P3Y8M
    The days component of this duration is 0.

  A sample xs:dayTimeDuration: P2DT4H23M12.2S
    The days component of this duration is 2.
```

Notice that calling `days-from-duration( )` against `xs:yearMonthDuration` returns `0`, as we mentioned earlier.

See Also

The definitions of the [2.0] `hours-from-duration( )`, [2.0] `minutes-from-duration( )`, [2.0] `months-from-duration( )`, [2.0] `seconds-from-duration( )`, and [2.0] `years-from-duration( )` functions.

[2.0] deep-equal()

Compares two sequences of items to see whether they and all their descendants are equal.

Syntax

```
xs:boolean deep-equal(item()*, item()*)
xs:boolean deep-equal(item()*, item()*, $collation as xs:string)
```

Inputs

Two sequences of items. An optional third argument identifies a collation that should be used when comparing string values. (The collation is not used when comparing node names, so `<strasse>` and `<straße>` are never equal, even though a German collation might find those string values to be the same.)

Output

`true` if the items are *deep-equal* to each other; `false` otherwise. Given the variety of types of items in XSLT 2.0, the rules for two sequences being deep-equal are somewhat complicated:

- If both values are empty sequences, `deep-equal( )` returns `true`.
- If the two values are sequences of different lengths (`count($value1) != count($value2)`), `deep-equal( )` returns `false`.
- If the two values are nodes of different kinds (an element and an attribute, for example), `deep-equal( )` returns `false`.
- If the two values are document nodes, they are equal only if `$value1/(*|text( ))` is deep-equal to `$value2/(*|text( ))`.
- If the two values are element nodes, they must have the same name and the same number of attributes. Also, those attributes must have the same values, and children must all be deep-equal.
- The remaining node types (attributes, processing instructions, comments, and text) are straightforward. For attributes and processing instructions, they must have the same name and the same values. For comments and text, their string values must be equal.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.3, "Equals, Union, Intersection and Except."

Example

We'll use our chocolate sales document to test the `deep-equal( )` function:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
```

```
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

Here is a stylesheet that compares several different values. Notice that `deep-equal( )` compares both atoms and nodes.

```
<?xml version="1.0"?>
<!-- deep-equal.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:variable name="testTree" as="node()*">
    <ora:report month="8" year="2006"
      xmlns:ora="http://www.oreilly.com">
      <ora:title>Chocolate bar sales</ora:title>
      <ora:brand>
        <ora:name>Lindt</ora:name>
        <ora:units>27408</ora:units>
      </ora:brand>
    </ora:report>

    <dw:report month="8" year="2006"
      xmlns:dw="http://www.oreilly.com">
      <dw:title>Chocolate bar sales</dw:title>
      <dw:brand>
        <dw:name>Lindt</dw:name>
        <dw:units>27408</dw:units>
      </dw:brand>
    </dw:report>

    <report month="8" year="2006">
      <title>Chocolate bar sales</title>
      <brand>
        <name>Lindt</name>
        <units>27408</units>
      </brand>
    </report>

    <address>
      <company>IBM Boeblingen</company>
```

```
    <street>Schoenaicherstrasse 220</street>
  </address>

  <address>
    <company>IBM B&#xF6;blingen</company>
    <street>Sch&#xF6;naicherstra&#xDF;e 220</street>
  </address>
</xsl:variable>

<xsl:variable name="sequence1" as="item()*">
  <xsl:sequence select="(3, 4, 5)"/>
</xsl:variable>

<xsl:variable name="sequence2" as="item()*">
  <xsl:sequence select="(3, 5, 4)"/>
</xsl:variable>

<xsl:variable name="sequence3" as="item()*">
  <xsl:sequence select="(3, 4, 5)"/>
</xsl:variable>

<xsl:template match="/">
  <xsl:text>&#xA;Tests of the deep-equal() function:</xsl:text>

  <xsl:text>&#xA;&#xA;  $sequence1 = (</xsl:text>
  <xsl:value-of select="$sequence1" separator=", "/>
  <xsl:text>)</xsl:text>
  <xsl:text>&#xA;  $sequence2 = (</xsl:text>
  <xsl:value-of select="$sequence2" separator=", "/>
  <xsl:text>)</xsl:text>
  <xsl:text>&#xA;  $sequence3 = (</xsl:text>
  <xsl:value-of select="$sequence3" separator=", "/>
  <xsl:text>)</xsl:text>

  <xsl:text>&#xA;&#xA;  deep-equal(</xsl:text>
  <xsl:text>$sequence1, $sequence2) = </xsl:text>
  <xsl:value-of select="deep-equal($sequence1, $sequence2)"/>

  <xsl:text>&#xA;&#xA;  deep-equal(</xsl:text>
  <xsl:text>$sequence1, $sequence3) = </xsl:text>
  <xsl:value-of select="deep-equal($sequence1, $sequence3)"/>

  <xsl:text>&#xA;&#xA;  deep-equal(</xsl:text>
  <xsl:text>subsequence($sequence1, 3, 1), &#xA;</xsl:text>
  <xsl:text>             subsequence($sequence2, 2, 1) = </xsl:text>
  <xsl:value-of
    select="deep-equal(subsequence($sequence1, 3, 1),
            subsequence($sequence2, 2, 1))"/>

  <xsl:text>&#xA;&#xA;  Comparing the first two </xsl:text>
  <xsl:text>subtrees in $testTree:</xsl:text>
  <xsl:text>&#xA;    deep-equal(subsequence($testTree, 1, 1),&#xA;</xsl:text>
  <xsl:text>               subsequence($testTree, 2, 1)) = </xsl:text>
  <xsl:value-of
    select="deep-equal(subsequence($testTree, 1, 1),
```

```
            subsequence($testTree, 2, 1))"/>

    <xsl:text>&#xA;&#xA;  Comparing part of our input </xsl:text>
    <xsl:text>document to &#xA;      part of the third subtree </xsl:text>
    <xsl:text>in $testTree:</xsl:text>
    <xsl:text>&#xA;    deep-equal(/report/brand[1]/units,$#xA;</xsl:text>
    <xsl:text>               subsequence($testTree, 3, 1)</xsl:text>
    <xsl:text>/brand[1]/units) = </xsl:text>
    <xsl:value-of
      select="deep-equal(/report/brand[1]/units,
            subsequence($testTree, 3, 1)/brand[1]/units)"/>

    <xsl:text>&#xA;&#xA;  Comparing two German addresses:</xsl:text>
    <xsl:text>&#xA;    deep-equal(subsequence($testTree, 4, 1),&#xA;</xsl:text>
    <xsl:text>               subsequence($testTree, 5, 1)) = </xsl:text>
    <xsl:value-of
      select="deep-equal(subsequence($testTree, 4, 1),
            subsequence($testTree, 5, 1))"/>

    <xsl:text>&#xA;&#xA;  Comparing two German addresses </xsl:text>
    <xsl:text>using German collation: </xsl:text>
    <xsl:text>&#xA;    deep-equal(subsequence($testTree, 4, 1),&#xA;</xsl:text>
    <xsl:text>               subsequence($testTree, 5, 1), &#xA;</xsl:text>
    <xsl:text>               $GermanCollation) = </xsl:text>
    <xsl:value-of
      select="deep-equal(subsequence($testTree, 4, 1),
            subsequence($testTree, 5, 1),
            concat('http://saxon.sf.net/collation?',
                  'class=com.oreilly.xslt.GermanCollation;'))"/>
  </xsl:template>

</xsl:stylesheet>
```

The results are:

```
Tests of the deep-equal() function:

  $sequence1 = (3, 4, 5)
  $sequence2 = (3, 5, 4)
  $sequence3 = (3, 4, 5)

  deep-equal($sequence1, $sequence2) = false

  deep-equal($sequence1, $sequence3) = true

  deep-equal(subsequence($sequence1, 3, 1),
             subsequence($sequence2, 2, 1) = true

  Comparing the first two subtrees in $testTree:
    deep-equal(subsequence($testTree, 1, 1),
               subsequence($testTree, 2, 1)) = true

  Comparing part of our input document to
      part of the third subtree in $testTree:
    deep-equal(/report/brand[1]/units,
               subsequence($testTree, 3, 1)/brand[1]/units) = true
```

```
Comparing two German addresses:
  deep-equal(subsequence($testTree, 4, 1),
             subsequence($testTree, 5, 1)) = false

Comparing two German addresses using German collation:
  deep-equal(subsequence($testTree, 4, 1),
             subsequence($testTree, 5, 1),
             $GermanCollation) = true
```

We start by creating a variable that contains three root nodes. The first two are identical except for different namespace prefixes. The third root node in `$testTree` is similar to part of our chocolate sales document. We also create three sequences of atomic values. We compare the sequences of atoms with each other, and we compare one item from `$sequence1` with one item from `$sequence2`.

Comparing the first two root nodes in `$testTree`, they are in fact deep-equal to each other because the namespace prefix doesn't matter. Both prefixes map to the same URI, so the two root nodes are the same. If the two nodes use the same prefixes, but those two identical prefixes map to two different URLs, those two nodes would not be deep-equal to each other.

Next, we compare the `<units>` element from the first `<brand>` element in the third root node in `$testTree` with the similar element from our input document. The two nodes are deep-equal to each other.

Finally, we use the two `<address>` elements in `$testTree`. Using the default collation, the text values `IBM Boeblingen` and `IBM Böblingen` are different, as are `Schoenaicherstrasse 220` and `Schönaicherstraße 220`. Using the German collation, those strings are the same, so `deep-equal( )` returns true. (To keep the listing within the margins of the page, we used the `concat( )` function to combine the two halves of the Saxon collation URI.) See "The document() Function and Sorting" in Chapter 8 for more information.

[2.0] default-collation()

Returns a string that represents the default collation.

Syntax

```
xs:string default-collation()
```

Inputs

None.

Output

An `xs:string` that represents the default collation.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 16, "Context Functions."

Example

Here's a stylesheet that simply displays the default collation:

```
<?xml version="1.0"?>
<!-- default-collation.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;The default collation is: &#xA;  </xsl:text>
    <xsl:value-of select="default-collation()"/>
  </xsl:template>

</xsl:stylesheet>
```

The results are:

```
The default collation is:
  http://www.w3.org/2005/xpath-functions/collation/codepoint
```

The default collation, as defined in the XQuery 1.0 and XPath 2.0 Functions and Operators spec, is the Unicode code point collation. That collation is associated with the URI `http://www.w3.org/2005/xpath-functions/collation/codepoint`, and all implementations of XPath 2.0 and XQuery 1.0 are required to support it. Implementors are free to support other collation schemes, and each processor can define its own mechanisms for associating a URI with a particular collation. Each processor can define its own default collation as well.

A final note: the default collation can be overridden in functions that specify a collation sequence, such as `compare( )`, `deep-equal( )`, `ends-with( )`, or `max( )`.

[2.0] distinct-values()

Given a sequence, returns a new sequence containing one copy of each unique value in the original sequence.

Syntax

```
xs:anyAtomicType* distinct-values(xs:anyAtomicType*)
xs:anyAtomicType* distinct-values(xs:anyAtomicType*, $collation as xs:string)
```

Inputs

A sequence of atomic values. `distinct-values( )` also has an optional argument specifying a collation algorithm. If present, the collation algorithm is used to determine whether two values are different from each other.

Output

A sequence containing only one copy of each atomic value in the original sequence. Some minor complications are as follows:

- When `distinct-values( )` returns its result sequence, the order in which the unique values are returned is implementation-defined. Also, within a group of values that have the same value, *which* value is returned is implementation-defined.
- If the input sequence is the empty sequence, the empty sequence is returned. Passing the empty sequence to `distinct-values( )` does not raise an error.
- When comparing `xs:float` and `xs:double` values, positive zero and negative zero are considered equal. Also, if there are numbers with the value `NaN` (not a number), only one of those values will be in the result sequence.
- When comparing `xs:date`, `xs:dateTime`, and `xs:time` values, if those values don't have a timezone component, their timezone is considered to be the timezone returned by the `implicit-timezone( )` function. Also be aware that `xs:date`, `xs:dateTime`, and `xs:time` values can be equal even though their timezone values are different. Noon in one timezone might be equal to 9 a.m. in another, for example.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.1, "General Functions and Operators on Sequences."

Example

We'll start with our simplified address book to illustrate the `distinct-values( )` function:

```
<?xml version="1.0"?>
<!-- simple-addresses.xml -->
<addressbook>
  <address>
    <name>Mr. Chester Hasbrouck Frisby</name>
    <city>Sheboygan</city>
    <state>WI</state>
  </address>
  <address>
    <name>Mary Backstayge</name>
    <city>Skunk Haven</city>
    <state>MA</state>
  </address>
  <address>
    <name>Ms. Natalie Attired</name>
    <city>Winter Harbor</city>
    <state>ME</state>
  </address>
  <address>
    <name>Harry Backstayge</name>
    <city>Skunk Haven</city>
    <state>MA</state>
  </address>
  <address>
    <name>Mary McGoon</name>
    <city>Boylston</city>
    <state>VA</state>
```

```
  </address>
  <address>
    <name>Ms. Amanda Reckonwith</name>
    <city>Lynn</city>
    <state>MA</state>
  </address>
</addressbook>
```

Here's a simple stylesheet that lists the distinct values of the `<state>` elements in the XML source document:

```
<?xml version="1.0"?>
<!-- distinct-values1.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Here is a test of the distinct-values() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>&#xA;  The states in the source </xsl:text>
    <xsl:text>document are:&#xA;    </xsl:text>
    <xsl:value-of
      select="/addressbook/address/state"
      separator=", "/>

    <xsl:text>&#xA;&#xA;  The unique states </xsl:text>
    <xsl:text>is:&#xA;    </xsl:text>
    <xsl:value-of
      select="distinct-values(/addressbook/address/state)"
      separator=", "/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Here is a test of the distinct-values() function:

  The states in the source document are:
    WI, MA, ME, MA, VA, MA

  The unique states is:
    WI, MA, ME, VA
```

In this example, the values returned by `distinct-values( )` are returned in document order, although other XSLT processors might return those values in sorted order or some other way.

Here's a stylesheet that compares some of the special cases we mentioned earlier:

```
<?xml version="1.0"?>
<!-- distinct-values2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xsl:output method="text"/>

    <xsl:template match="/">
      <xsl:text>&#xA;Here are some tests of the </xsl:text>
      <xsl:text>distinct-values() </xsl:text>
      <xsl:text>function:&#xA;</xsl:text>

      <xsl:text>&#xA;  A sequence of xs:time values:</xsl:text>
      <xsl:variable name="timeSequence" as="xs:time*">
        <xsl:sequence
          select="(xs:time('00:43:00-06:00'),
                   xs:time('01:43:00-05:00'),
                   xs:time('02:43:00'),
                   xs:time('07:35:00'),
                   current-time())"/>
      </xsl:variable>

      <xsl:text>&#xA;&#xA;    </xsl:text>
      <xsl:value-of
        select="distinct-values($timeSequence)"
        separator="&#xA;    "/>

      <xsl:text>&#xA;&#xA;  A sequence of integer values:</xsl:text>
      <xsl:variable name="numberSequence" as="xs:integer*">
        <xsl:sequence select="(3, 8, (2 + 1), -0, 0)"/>
      </xsl:variable>

      <xsl:text>&#xA;    </xsl:text>
      <xsl:value-of
        select="distinct-values($numberSequence)"
        separator=", "/>
    </xsl:template>

</xsl:stylesheet>
```

The results from this stylesheet look like this:

```
Here are some tests of the distinct-values() function:

  A sequence of xs:time values:

    00:43:00-06:00
    07:35:00
    06:23:05.19-04:00

  A sequence of integer values:
    3, 8, 0
```

In the first sequence, we compare five different `xs:time` values. The first three are actually equal; they represent the same time in three different zones. Notice that the third `xs:time` value uses the implicit timezone; you'll get different results if you run this stylesheet on a machine that isn't set to the `GMT-4` timezone. The fourth and fifth values are different, so there

are three values in the output sequence. We used Saxon for this example; for duplicate values, it returns the first one it finds.

For the second sequence, we have several numeric values. The values `3` and `(2 + 1)` are identical, and `-0` and `0` are the same as well. Notice that in this case the value in the output sequence is the second zero value, not the first. (Saxon normalizes negative zero to zero, although this is implementation-defined.)

As a final example, we'll use a collation function. Here is the stylesheet:

```
<?xml version="1.0"?>
<!-- distinct-values3.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A final test of the </xsl:text>
    <xsl:text>distinct-values() function:&#xA;&#xA;</xsl:text>

    <xsl:variable name="testStrings" as="xs:string*">
      <xsl:sequence
        select="'street', 'Strasse', 'Stra&#xDF;e'"/>
    </xsl:variable>

    <xsl:text>  The test sequence: &#xA;    </xsl:text>
    <xsl:value-of select="$testStrings" separator=", "/>
    <xsl:text>&#xA;  distinct-values($testStrings):</xsl:text>
    <xsl:text>&#xA;    </xsl:text>
    <xsl:value-of select="distinct-values($testStrings)"
      separator=", "/>

    <xsl:text>&#xA;  distinct-values($testStrings, </xsl:text>
    <xsl:text>[German collation]):&#xA;    </xsl:text>
    <xsl:value-of
      select="distinct-values($testStrings,
              concat('http://saxon.sf.net/collation?',
                     'class=com.oreilly.xslt.GermanCollation;'))"
      separator=", "/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
A final test of the distinct-values() function:

  The test sequence:
    street, Strasse, Straße
  distinct-values($testStrings):
    street, Strasse, Straße
  distinct-values($testStrings, [German collation]):
    street, Strasse
```

The German sharp-s character (ß) is equivalent to `ss`. The default collation doesn't recognize this, but our custom collation does. (To keep the listing within the margins of the page, we used the `concat( )` function to combine the two halves of the Saxon collation URI.) See "The document() Function and Sorting" in Chapter 8 for more information.

[2.0] doc()

Given a URI, returns a document node representing the contents of that URI.

Syntax

```
document-node()? doc($uri as xs:string?)
```

Inputs

A URI. The format of the URI is implementation-defined.

Output

A document node that represents the contents of the requested URI.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.5, "Functions and Operators that Generate Sequences."

Example

Here is an example that uses Saxon's support for simple filenames as URIs. We'll invoke the `doc( )` function to load this XML document:

```
<?xml version="1.0"?>
<!-- polist.xml -->
<collection>
  <doc href="po38292.xml"/>
  <doc href="po38293.xml"/>
  <doc href="po38294.xml"/>
  <doc href="po38295.xml"/>
</collection>
```

Here's our stylesheet:

```
<?xml version="1.0"?>
<!-- doc.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the doc() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  Here are all the purchase orders </xsl:text>
    <xsl:text>listed in polist.xml:</xsl:text>
    <xsl:for-each select="doc('polist.xml')/collection/doc">
```

```
      <xsl:text>&#xA;    </xsl:text>
      <xsl:value-of select="@href"/>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

We use the `doc( )` function to load the file *polist.xml*, and then we use `<xsl:for-each>` to process the contents of the document we just loaded. Here are the results:

```
A test of the doc() function:

  Here are all the purchase orders listed in polist.xml:
    po38292.xml
    po38293.xml
    po38294.xml
    po38295.xml
```

Notice that we use the `doc( )` function in an XPath expression. We iterate through all the `<doc>` elements in the source file using the document root returned by the `doc( )` function. We could also use the `doc( )` function to open each of the documents referenced in the *polist.xml* file.

See Also

The [2.0] `collection( )`, [2.0] `doc-available( )`, and `document( )` functions and the discussion of "[2.0] The doc() and doc-available() Functions" in Chapter 8.

[2.0] doc-available()

Tests to see if a given document node is available.

Syntax

```
xs:boolean doc-available($uri as xs:string)
```

Inputs

The URI of a document.

Output

This function returns `true` if the `doc( )` function can successfully load the requested URI. If the argument cannot be converted to an `xs:anyURI`, the XSLT processor raises an error. Otherwise, `doc-available( )` returns `false`. The format of the URI is implementation-defined.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.5, "Functions and Operators that Generate Sequences."

Example

Here's a short stylesheet that tests whether two documents are available. This uses Saxon's support for simple filenames as URIs:

```
<?xml version="1.0"?>
<!-- doc-available.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the doc-available() function:&#xA;</xsl:text>

    <xsl:text>&#xA;  doc-available('polist.xml') = </xsl:text>
    <xsl:value-of select="doc-available('polist.xml')"/>

    <xsl:text>&#xA;&#xA;  doc-available('polist2.xml') = </xsl:text>
    <xsl:value-of select="doc-available('polist2.xml')"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Tests of the doc-available() function:

  doc-available('polist.xml') = true

  doc-available('polist2.xml') = false
```

The file *polist.xml* exists in the current directory and can be loaded by Saxon; the file *polist2.xml* does not exist in the current directory.

Many of the details of the `doc-available( )` function are implementation-defined. See the documentation for your XSLT processor to see how it implements `doc-available( )` and what URI formats it supports.

document()

Allows you to process multiple source documents in a single stylesheet. This extremely powerful and flexible function is discussed extensively in Chapter 7, so we'll only include a brief overview of the function here.

Syntax

```
[1.0] node-set document(object, node-set?)
[2.0] node()* document(item()*)
[2.0] node()* document(item()*, node())
```

Inputs

The `document( )` function most commonly takes a string as its argument; that string is treated as a URI, and the XSLT processor attempts to open that URI and parse it. If the string is empty

(the function call is `document('')`), the `document( )` function parses the stylesheet itself. See "Grouping Nodes" in Chapter 7 for all the details on the parameters to the `document( )` function.

[2.0] In XSLT 2.0, the `document( )` function can also take a second argument, a node used to find the base URI property of the requested documents. The base URI of the node is combined with the resource names in the first argument to form a complete URI.

Output

A *[1.0]* node-set or *[2.0]* sequence containing the nodes identified by the input argument. Again, Chapter 7 has all the details, so we won't rehash them here.

Defined in

[1.0] XSLT section 12.1, "Multiple Source Documents."

[2.0] XSLT section 16.1, "Multiple Source Documents."

Example

The following example uses the `document( )` function with an empty string to implement a lookup table. Here is our XML document:

```
<?xml version="1.0"?>
<!-- polist.xml -->
<collection>
  <doc href="po38292.xml"/>
  <doc href="po38293.xml"/>
  <doc href="po38294.xml"/>
  <doc href="po38295.xml"/>
</collection>
```

We'll use the `document( )` function against each `href` attribute in this XML file. Calling `document( )` returns a set of nodes representing the referenced document. Each `href` points to a purchase order document that is something like this:

```
<?xml version="1.0" ?>
<!-- po38293.xml -->
<purchase-order id="38293">
  <date year="2001" month="9" day="8"/>
  <customer id="4738" level="Basic">
    <address type="business">
      <name>
        <title>Ms.</title>
        <first-name>Amanda</first-name>
        <last-name>Reckonwith</last-name>
      </name>
      <street>930-A Chestnut Street</street>
      <city>Lynn</city>
      <state>MA</state>
      <zip>02930</zip>
    </address>
    <address type="ship-to"/>
  </customer>
  <items>
```

```
    <item part-no="23813-03-CDK">
      <name>Cucumber Decorating Kit</name>
      <qty>1</qty>
      <price>29.95</price>
    </item>
  </items>
</purchase-order>
```

We want our stylesheet to find all of the purchase orders in which a customer ordered more than one item. Here's how the stylesheet looks:

```
<?xml version="1.0"?>
<!-- document.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the document() function:&#xA;</xsl:text>

    <xsl:for-each select="/collection/doc">
      <xsl:variable name="latestDoc" select="document(@href)"/>
      <xsl:if
        test="count($latestDoc/purchase-order/items/item) &gt; 1">
        <xsl:text>&#xA;  </xsl:text>
        <xsl:value-of
          select="$latestDoc/purchase-order/customer/
                  address/name/first-name"/>
        <xsl:text> </xsl:text>
        <xsl:value-of
          select="$latestDoc/purchase-order/customer/
                  address/name/last-name"/>
        <xsl:text>&#xA;    ordered </xsl:text>
        <xsl:value-of
          select="count($latestDoc/purchase-order/items/item)"/>
        <xsl:text> items on </xsl:text>
        <xsl:value-of
          select="$latestDoc/purchase-order/date/@month"/>
        <xsl:text>/</xsl:text>
        <xsl:value-of
          select="$latestDoc/purchase-order/date/@day"/>
        <xsl:text>/</xsl:text>
        <xsl:value-of
          select="$latestDoc/purchase-order/date/@year"/>
        <xsl:text> - see P.O. #</xsl:text>
        <xsl:value-of
          select="$latestDoc/purchase-order/@id"/>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Running the stylesheet gives us these results:

```
A test of the document() function:

  Chester Hasbrouck Frisby
    ordered 3 items on 6/19/2001 - see P.O. #38292
  Natalie Attired
    ordered 2 items on 4/21/2001 - see P.O. #38294
```

The stylesheet has used the `document( )` function to open all of the purchase orders referenced in *polist.xml*. For any orders in which the customer ordered more than one item (`count($latestDoc/purchase-order/items/item) > 1`), we print out a few details from the purchase order. Notice that we re-initialize the variable `$latestDoc` as we iterate through all of the `href` attributes; if we find a document that we want to process, storing the tree in a variable means we don't have to call `document( )` over and over to get the information we need.

[2.0] document-uri()

Given a node, returns the document URI property for that node.

Syntax

```
xs:anyURI? document-uri(node()?)
```

Inputs

A node.

Output

If the node is not a document node, `document-uri( )` returns the empty sequence. Otherwise, it returns the document URI property for the node. If the argument is the empty sequence, `document-uri( )` returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 2, "Accessors."

Example

Here's a short stylesheet that returns the document URI for the XML document we're processing:

```
<?xml version="1.0"?>
<!-- document-uri.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the document-uri() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  The document URI for the </xsl:text>
```

```
    <xsl:text>root element is:&#xA;    </xsl:text>
    <xsl:value-of select="document-uri(/)"/>
  </xsl:template>

</xsl:stylesheet>
```

When we run this stylesheet against the file *blank.xml*, here are the results:

```
A test of the document-uri() function:

  The document URI for the root element is:
    file:/C:/projects/XSLTbookV2/AppendixC/blank.xml
```

element-available()

Determines if a given element is available to the XSLT processor. This function allows you to design stylesheets that react gracefully if a particular extension element is not available to process an XML document.

Syntax

```
[1.0] boolean element-available($elementName as string)
[2.0] xs:boolean element-available($elementName as xs:string)
```

Inputs

The element's name. The name should be qualified with a namespace; if the namespace URI is the same as the XSLT namespace URI, then the element name refers to an element defined by XSLT. Otherwise, the name refers to an extension element. If the element name has a null namespace URI, then the `element-available` function returns `false`. *[2.0]* If the argument cannot be converted to a valid QName, the XSLT processor raises an error.

Output

The boolean value `true` if the element is available; `false` otherwise.

Defined in

[1.0] XSLT section 15, "Fallback."

[2.0] XSLT section 18.2, "Extension Instructions."

Example

We'll illustrate the `element-available( )` function with the following XML document:

```
<?xml version="1.0"?>
<!-- chapterlist.xml -->
<book>
  <title>XSLT</title>
  <chapter>
    <title>Getting Started</title>
    <para>If this chapter had any text, it would appear here.</para>
  </chapter>
  <chapter>
```

```
    <title>The Hello World Example</title>
    <para>If this chapter had any text, it would appear here.</para>
  </chapter>
  ...
  <chapter>
    <title>Combining XML Documents</title>
    <para>If this chapter had any text, it would appear here.</para>
  </chapter>
</book>
```

Our stylesheet attempts to use the Xalan-J `<redirect:write>` to write each `<chapter>` to a different file. If that element is not available, we write all of the information to a single HTML file. Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- element-available.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:redirect="org.apache.xalan.xslt.extensions.Redirect"
  extension-element-prefixes="redirect">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <xsl:choose>
      <xsl:when test="element-available('redirect:write')">
        <xsl:for-each select="/book/chapter">
          <redirect:write select="concat('chapter', position(), '.html')">
            <html>
              <head>
                <title><xsl:value-of select="title"/></title>
              </head>
              <body>
                <h1><xsl:value-of select="title"/></h1>
                <xsl:apply-templates select="para"/>
                <xsl:if test="not(position()=1)">
                  <p>
                    <a href="chapter{position()-1}.html">Previous</a>
                  </p>
                </xsl:if>
                <xsl:if test="not(position()=last())">
                  <p>
                    <a href="chapter{position()+1}.html">Next</a>
                  </p>
                </xsl:if>
              </body>
            </html>
          </redirect:write>
        </xsl:for-each>
      </xsl:when>
      <xsl:otherwise>
        <html>
          <head>
            <title><xsl:value-of select="/book/title"/></title>
          </head>
          <xsl:for-each select="/book/chapter">
```

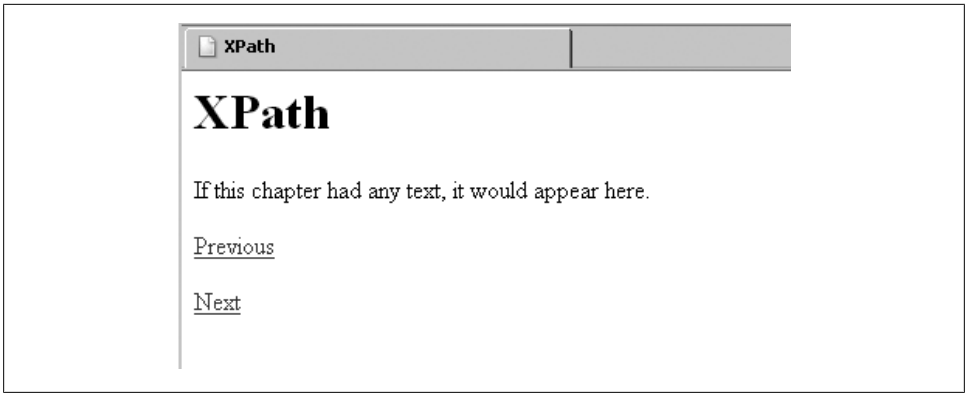


Figure C-1. Sample HTML output file

```
            <h1><xsl:value-of select="title"/></h1>
            <xsl:apply-templates select="para"/>
          </xsl:for-each>
        </html>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

  <xsl:template match="para">
    <p><xsl:apply-templates select="*|text()"/></p>
  </xsl:template>

</xsl:stylesheet>
```

If the `<redirect:write>` element is available, we'll get a separate file for each `<chapter>` element. Those files look like this:

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
      <title>XPath</title>
   </head>
   <body>
      <h1>XPath</h1>
      <p>If this chapter had any text, it would appear here.</p>
      <p><a href="chapter2.html">Previous</a></p>
      <p><a href="chapter4.html">Next</a></p>
   </body>
</html>
```

When rendered in a browser, the file looks like Figure C-1.

Clicking on the Previous link takes you to the file *chapter2.html*, while clicking on the Next link takes you to *chapter4.html*.

Although the format of the message is slightly different, the output in the multiple HTML files is the same.

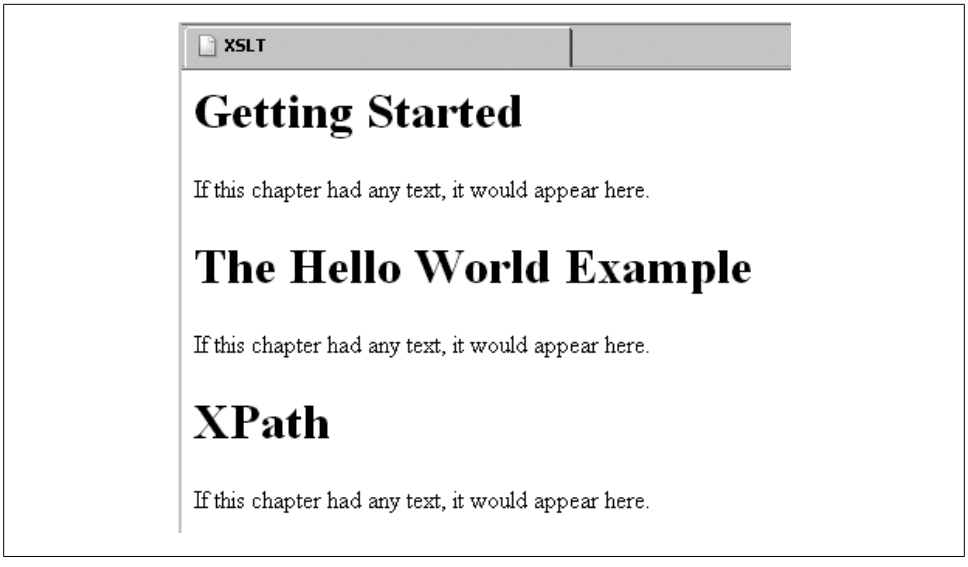

XSLT
Getting Started
If this chapter had any text, it would appear here.
The Hello World Example
If this chapter had any text, it would appear here.
XPath
If this chapter had any text, it would appear here.

Figure C-2. HTML document listing all chapters

If we use Saxon, which doesn't have the `<redirect:write>` element, we get a single HTML file that looks like this:

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>XSLT</title>
   </head>
   <h1>Getting Started</h1>
   <p>If this chapter had any text, it would appear here.</p>
   ...
   <h1>Combining XML Documents</h1>
   <p>If this chapter had any text, it would appear here.</p>
</html>
```

When rendered, our output looks like Figure C-2.

MSXSL and the AltovaXML engine generate the same results.

In this example, the `element-available( )` function allows us to determine what processing capabilities are available and to respond gracefully to whatever we find.

[2.0] empty()

Given a sequence, returns `true` if the argument is the empty sequence, `false` otherwise.

Syntax

```
xs:boolean empty(item()*)
```

Inputs

A sequence of items.

Output

`true` if the argument is the empty sequence; `false` otherwise. Although `empty($sequence)` and `count($sequence) = 0` are logically equivalent, `empty( )` is likely to be more efficient.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.1, "General Functions and Operators on Sequences."

Example

This stylesheet illustrates the `empty( )` function:

```
<?xml version="1.0"?>
<!-- empty.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:variable name="emptySequence" as="item()*">
      <xsl:sequence select="()"/>
    </xsl:variable>

    <xsl:variable name="singleton" as="item()*">
      <xsl:sequence select="(3)"/>
    </xsl:variable>

    <xsl:variable name="longSequence" as="item()*">
      <xsl:sequence
        select="(3, 4, 5, current-date(), current-time(), 8, 'blue',
                 'red', xs:float(3.14), 42, xs:date('1995-04-21'))"/>
    </xsl:variable>

    <xsl:text>&#xA;Here are some tests of the empty() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>&#xA;  Our first sequence is:&#xA;&#xA;    </xsl:text>
    <xsl:value-of select="$emptySequence" separator="&#xA;    "/>
    <xsl:text>&#xA;&#xA;</xsl:text>
    <xsl:value-of select="if (empty($emptySequence))
                          then 'This is the empty sequence!'
                          else 'This is not the empty sequence!'"/>
    <xsl:text>&#xA;&#xA;</xsl:text>

    <xsl:text>&#xA;  Our next sequence is:&#xA;&#xA;    </xsl:text>
    <xsl:value-of select="$singleton" separator="&#xA;    "/>
    <xsl:text>&#xA;&#xA;</xsl:text>
```

```
    <xsl:value-of select="if (empty($singleton))
                          then 'This is the empty sequence!'
                          else 'This is not the empty sequence!'"/>
    <xsl:text>&#xA;&#xA;</xsl:text>

    <xsl:text>&#xA;  Our final sequence is:&#xA;&#xA;    </xsl:text>
    <xsl:value-of select="$longSequence" separator="&#xA;    "/>
    <xsl:text>&#xA;&#xA;</xsl:text>
    <xsl:value-of select="if (empty($longSequence))
                          then 'This is the empty sequence!'
                          else 'This is not the empty sequence!'"/>
    <xsl:text>&#xA;&#xA;</xsl:text>

  </xsl:template>

</xsl:stylesheet>
```

The stylesheet generates these results:

```
Here are some tests of the empty() function:

  Our first sequence is:


This is the empty sequence!


  Our next sequence is:
    3

This is not the empty sequence!


  Our final sequence is:
    3
    4
    5
    2008-03-04-05:00
    11:04:14.25-05:00
    8
    blue
    red
    3.14
    42
    1995-04-21

This is not the empty sequence!
```

[2.0] encode-for-uri()

Given a string, encodes that string for use in a URI.

Syntax

```
xs:string encode-for-uri($uri-part as xs:string?)
```

Input

A string containing the value to be escaped.

Output

The input string with the appropriate characters escaped as percent-encoded values. All characters are escaped *except* the following: upper- and lowercase letters `A–Z`, the digits `0–9`, the hyphen-minus character ("`-`" or `-`), the underscore or low-line character ("`_`" or `_`), the period or full stop ("`.`" or `&#2E;`), and the tilde ("~" or `~`). For example, the space character, `&#x20`, is escaped as `%20`. All hexadecimal digits are in uppercase, so the character `á` (á, a lowercase `a` with an acute accent) is escaped as `%E1`.

Keep in mind that the `encode-for-uri( )` function escapes every character, so you should use it to escape only the parts of a URI that need to be escaped. For example, calling `encode-for-uri('http://www.oreilly.com/')` returns `http%3A%2F%2Fwww.oreilly.com%2F`, which is almost certainly *not* what you want.

If the value of the argument is the empty sequence, a zero-length string is returned.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 7.4, "Functions on String Values."

Example

Here's a stylesheet that features two examples courtesy of the XQuery 1.0 and XPath 2.0 Functions and Operators spec:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- encode-for-uri.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the encode-for-uri() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  encode-for-uri</xsl:text>
    <xsl:text>('http://www.oreilly.com/') = &#xA;    </xsl:text>
    <xsl:value-of
      select="encode-for-uri('http://www.oreilly.com/')"/>

    <xsl:text>&#xA;&#xA;  concat('http://www.example.com/', </xsl:text>
    <xsl:text>&#xA;          encode-for-uri</xsl:text>
    <xsl:text>('~b&#xE9;b&#xE9;')) = &#xA;    </xsl:text>
    <xsl:value-of
      select="concat('http://www.example.com/',
              encode-for-uri('~b&#xE9;b&#xE9;'))"/>

    <xsl:text>&#xA;&#xA;  concat('http://www.example.com/', </xsl:text>
    <xsl:text>&#xA;          encode-for-uri</xsl:text>
    <xsl:text>('100% organic')) = &#xA;    </xsl:text>
    <xsl:value-of
```

```
        select="concat('http://www.example.com/',
                encode-for-uri('100% organic'))"/>
    </xsl:template>

</xsl:stylesheet>
```

The stylesheet generates these results:

```
Tests of the encode-for-uri() function:

  encode-for-uri('http://www.oreilly.com/') =
    http%3A%2F%2Fwww.oreilly.com%2F

  concat('http://www.example.com/',
         encode-for-uri('~bébé')) =
    http://www.example.com/~b%C3%A9b%C3%A9

  concat('http://www.example.com/',
         encode-for-uri('100% organic')) =
    http://www.example.com/100%25%20organic
```

See Also

The descriptions of the [2.0] `escape-html-uri( )` and [2.0] `iri-to-uri( )` functions.

[2.0] ends-with()

Given two strings and an optional collation, returns `true` if the first string ends with the characters of the second string.

Syntax

```
xs:boolean ends-with(xs:string?, xs:string?)
xs:boolean ends-with(xs:string?, xs:string?, $collation as xs:string)
```

Inputs

Two strings and an optional name of a collation algorithm. A collation algorithm defines how characters are compared; to take an example from the specs, a collation algorithm might define that the characters `ss` and the German ß (sharp-s) character are the same.

Output

Assuming both arguments are not zero-length strings, if the first string ends with the second, `ends-with( )` returns the boolean value `true`; otherwise, it returns `false`. If the second string is a zero-length string, `ends-with( )` returns `true`. If the first string *and* the second string are both zero-length strings, `ends-with( )` returns true. If the first string is a zero-length string but the second string is not, `ends-with( )` returns `false`.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 7.5, "Functions Based on Substring Matching."

Example

Here's a short stylesheet that demonstrates how the `ends-with( )` function works:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- ends-with1.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="string1" select="'Have a nice day'"/>
    <xsl:variable name="string2" select="'Have a nice day!'"/>
    <xsl:variable name="string3" select="'day'"/>

    <xsl:text>&#xA;A test of the ends-with() function:&#xA;</xsl:text>

    <xsl:text>  ends-with('</xsl:text>
    <xsl:value-of select="$string1"/>
    <xsl:text>', '</xsl:text>
    <xsl:value-of select="$string3"/>
    <xsl:text>') = </xsl:text>
    <xsl:value-of select="ends-with($string1, $string3)"/>
    <xsl:text>&#xA;</xsl:text>

    <xsl:text>  ends-with('</xsl:text>
    <xsl:value-of select="$string2"/>
    <xsl:text>', '</xsl:text>
    <xsl:value-of select="$string3"/>
    <xsl:text>') = </xsl:text>
    <xsl:value-of select="ends-with($string2, $string3)"/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
A test of the ends-with() function:
  ends-with('Have a nice day', 'day') = true
  ends-with('Have a nice day!', 'day') = false
```

The `ends-with` function is one of several that can contain an optional collation. How that collation is specified varies from one processor to the next. In Saxon this is done with a URI that includes the name of the Java class that performs the collation. Here's an example that compares the German words *Straße* and *Strasse*. In the default collation, these two words are different; using the German collation, they're the same. Here's the stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- ends-with2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
```

```
    <xsl:variable name="string1" select="'Stra&#xDF;e'"/>
    <xsl:variable name="string2" select="'Strasse'"/>

    <xsl:text>&#xA;A test of the ends-with() function:&#xA;</xsl:text>

    <xsl:text>  ends-with('</xsl:text>
    <xsl:value-of select="$string1"/>
    <xsl:text>', '</xsl:text>
    <xsl:value-of select="$string2"/>
    <xsl:text>') = </xsl:text>
    <xsl:value-of select="ends-with($string1, $string2)"/>
    <xsl:text>&#xA;</xsl:text>

    <xsl:text>  ends-with('</xsl:text>
    <xsl:value-of select="$string1"/>
    <xsl:text>', '</xsl:text>
    <xsl:value-of select="$string2"/>
    <xsl:text>', [German collation]) = </xsl:text>
    <xsl:value-of
      select="ends-with($string1, $string2,
              concat('http://saxon.sf.net/collation?',
                     'class=com.oreilly.xslt.GermanCollation;'))"/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

And here are the results:

```
A test of the ends-with() function:
  ends-with('Straße', 'Strasse') = false
  ends-with('Straße', 'Strasse', [German collation]) = true
```

Using the collation we defined, the two spellings of `Strasse` are the same; using the default collation, they're not. (To keep the listing within the margins of the page, we used the `concat( )` function to combine the two halves of the Saxon collation URI.) See "The document() Function and Sorting" in Chapter 8 for more information.

[2.0] error()

Raises an error. This is equivalent to throwing an exception in Java, C++, Ruby, and other languages.

Syntax

```
none error()
none error($error as xs:QName)
none error($error as xs:QName?, $description as xs:string)
none error($error as xs:QName?, $description as xs:string,
           $error-object as item()*)
```

Inputs

This function has four different signatures. The three optional parameters defined in the "Syntax" section are the `QName` associated with the error, an `xs:string`, and a sequence of

items. How these inputs are processed is implementation-defined, so useful or appropriate values for these arguments vary from one processor to the next.

Output

This function never returns. The `error( )` function returns an error to the external processing environment. How (or if) the QName, string, and sequence of items passed to this function are delivered to the environment is implementation-defined.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 3, "The Error Function." In addition, Appendix C of this spec defines a number of error conditions and codes. For example, `err:FODT0003` is the error code for a timezone value that is not valid. The `err` namespace is bound to the URI `http://www.w3.org/2005/xqt-errors`—we'll use this in our examples. You're free to create your own error codes and messages if you like.

Example

We'll look at four stylesheets that throw errors, one for each of the four method signatures of the `error( )` function. Here's the first, which we simply call `error( )`:

```
<?xml version="1.0"?>
<!-- error1.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Test #1 of the error() function:</xsl:text>

    <xsl:value-of select="error()"/>
  </xsl:template>

</xsl:stylesheet>
```

Our results look like this:

```
    FOER0000: Error signalled by application call on error()
Transformation failed: Run-time errors were reported
```

The error code here, `FOER0000`, is defined in the XQuery 1.0 and XPath 2.0 Functions and Operators spec as "Unidentified error."

For our second attempt, we use a QName that refers to a specific error code from the spec:

```
<?xml version="1.0"?>
<!-- error2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Test #2 of the error() function:</xsl:text>
```

```
    <xsl:value-of
      select="error(QName('http://www.w3.org/2005/xqt-errors',
              'err:FORX0004'))"/>
  </xsl:template>

</xsl:stylesheet>
```

Our results:

```
  FORX0004: Error signalled by application call on error()
Transformation failed: Run-time errors were reported
```

These results have a specific error code, `FORX0004`. We'll move on to our third example, in which we add a descriptive string to the call to `error( )`:

```
<?xml version="1.0"?>
<!-- error3.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Test #3 of the error() function:</xsl:text>

    <xsl:value-of
      select="error(QName('http://www.w3.org/2005/xqt-errors',
              'err:FORX0004'),
              'Invalid replacement string.')"/>
  </xsl:template>

</xsl:stylesheet>
```

The results here are more informative; the error message includes our string:

```
  FORX0004: Invalid replacement string.
Transformation failed: Run-time errors were reported
```

The specs define error code `FORX0004` as "Invalid replacement string," so that's what we put into our error string. With Saxon, whatever string we pass to the `error( )` function is put into the output, so we're free to put more information in the string if we want.

Our final example includes an item—in this case, an `xs:dateTime` value:

```
<?xml version="1.0"?>
<!-- error4.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Test #4 of the error() function:</xsl:text>

    <xsl:value-of
      select="error(QName('http://www.w3.org/2005/xqt-errors',
              'err:FORX0004'),
```

```
                'Invalid replacement string.',
                current-dateTime())"/>
    </xsl:template>

</xsl:stylesheet>
```

Here our results are exactly what they were before:

```
    FORX0004: Invalid replacement string.
Transformation failed: Run-time errors were reported
```

Looking into the Saxon documentation, it seems the sequence of items we can pass to the `error( )` function are intended for more advanced techniques than simply writing messages to the console. The `saxon:try( )` function, available in the schema-aware version of Saxon, appears to have access to this data. Again, virtually all of the details of how the `error( )` function work are implementation-defined, so check your processor's documentation for more information.

[2.0] escape-html-uri()

Given an HTML URI, returns that URI with its non-ASCII characters escaped as UTF-8 characters.

Syntax

```
xs:string escape-html-uri(xs:string?)
```

Inputs

An `xs:string` representing an HTML URI.

Output

The input string with all characters outside the range ` ` to `~` escaped as UTF-8 characters. When a character is converted to UTF-8, it is represented as octets in the form `%HH`, where `HH` is the hexadecimal representation of the octet. The hexadecimal characters generated by `escape-html-uri( )` are in uppercase. This function conforms to the rules for escaping non-ASCII characters defined in the HTML 4.0 spec.

If argument is the empty sequence, a zero-length string is returned.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 7.4, "Functions on String Values."

Example

This stylesheet uses `escape-html-uri( )` to correctly encode characters in the URI (in this case `é`, a lowercase e with an acute accent) as UTF-8. (This example was taken from the XQuery 1.0 and XPath 2.0 Functions and Operators spec.)

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- escape-html-uri.xsl -->
<xsl:stylesheet version="1.0"
```

```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="text"/>

    <xsl:template match="/">
      <xsl:text>&#xA;Tests of the escape-html-uri() function:</xsl:text>

      <xsl:text>&#xA;&#xA;  escape-html-uri</xsl:text>
      <xsl:text>('http://www.example.com/~bébé') = &#xA;    </xsl:text>
      <xsl:value-of
        select="escape-html-uri('http://www.example.com/~bébé')"/>
    </xsl:template>

  </xsl:stylesheet>
```

The results look like this:

```
Tests of the escape-html-uri() function:

  escape-html-uri('http://www.example.com/~bébé') =
    http://www.example.com/~b%C3%A9b%C3%A9
```

Notice that the single-byte character in the original URL (é) has been replaced by a two-byte UTF-8 character (`%C3%A9`) in the output.

See Also

The descriptions of the [2.0] `encode-for-uri( )` and [2.0] `iri-to-uri( )` functions.

[2.0] exactly-one()

Raises an error unless its argument is a singleton, a sequence containing exactly one item. Be aware that `exactly-one( )` terminates processing. For a more flexible approach, use the `count( )` function to determine the cardinality of a sequence.

Syntax

```
item()+ exactly-one(item()*)
```

Inputs

A sequence.

Outputs

If the input sequence contains exactly one item, the input sequence is returned. Otherwise, `exactly-one( )` raises an error.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.2, "Functions That Test the Cardinality of Sequences." More details about this function can be found in XQuery 1.0 and XPath 2.0 Formal Semantics section 7.2, "Standard Functions with Specific Static Typing Rules."

Example

The exactly-one( ) function is useful for working with XSLT 2.0's static typing. Any sequence can be empty, a singleton, or have multiple items. If we need to test the cardinality of the sequence at runtime, the exactly-one( ) function can enforce the restriction that a sequence must be a singleton. However, exactly-one( ) terminates stylesheet processing if its argument doesn't have exactly one item, so it's far more flexible to use the count( ) function to check the number of items in a sequence before working with it.

Here is a short stylesheet that invokes the exactly-one( ) function with a singleton:

```
<?xml version="1.0"?>
<!-- exactly-one.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:text>&#xA;Here is a test of the exactly-one() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:variable name="singleton" as="item()*">
      <xsl:sequence select="0"/>
    </xsl:variable>

    <xsl:text>&#xA;  Our sequence is:&#xA;&#xA;    </xsl:text>
    <xsl:value-of select="$singleton" separator="&#xA;    "/>

    <xsl:if test="count(exactly-one($singleton))">
      <xsl:text>&#xA;&#xA;  Our test sequence has exactly </xsl:text>
      <xsl:text>one item!&#xA;</xsl:text>
    </xsl:if>
  </xsl:template>

</xsl:stylesheet>
```

In the test attribute of the <xsl:if> element we need to use exactly-one( ) to generate a boolean value. Here's a simple, yet incorrect, way to do this:

```
<xsl:if test="exactly-one($singleton)"> <!-- doesn't work! -->
```

If the exactly-one( ) function doesn't raise an error, it returns a sequence with exactly one item. It would seem that a one-item sequence would always be true, but that's not the case. In our example here, the singleton sequence (0) evaluates to false because the atomic value 0 is false. Using the count( ) function ensures that we get a meaningful result. (With a singleton sequence, count( ) always returns 1, which is always true.)

Also keep in mind that many datatypes cannot be converted to boolean values. If our singleton were the sequence (xs:date('1995-04-21')), the XSLT processor would raise an error because xs:date values cannot be converted to boolean.

At any rate, here are the results:

```
Here is a test of the exactly-one() function:

  Our sequence is:

    0

  Our test sequence has exactly one item!
```

See Also

The descriptions of the `count( )`, [2.0] `empty( )`, [2.0] `one-or-more( )`, and the [2.0] `zero-or-one( )` functions and the XPath 2.0 `treat as` operater (in the section "[2.0] Datatype Operators —instance of, castable as, cast as, and treat as" in Chapter 3).

[2.0] exists()

Returns `true` if the input sequence is nonempty; `false` otherwise.

Syntax

```
xs:boolean exists(item()*)
```

Input

A sequence of items.

Output

If the sequence of items is not empty, `exists( )` returns `true`; otherwise, it returns `false`.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.1, "General Functions and Operators on Sequences."

Example

This simple stylesheet uses the `exists( )` function against three sequences. The first sequence is the empty sequence, the second is a singleton, and the third is a longer sequence:

```
<?xml version="1.0"?>
<!-- exists.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:text>&#xA;Here is a test of the exists() </xsl:text>
    <xsl:text>function:&#xA;&#xA;  </xsl:text>

    <xsl:variable name="emptySequence" as="item()*">
```

```
    <xsl:sequence select="()"/>
  </xsl:variable>

  <xsl:variable name="singleton" as="item()*">
    <xsl:sequence select="(3)"/>
  </xsl:variable>

  <xsl:variable name="longSequence" as="item()*">
    <xsl:sequence
      select="(3, 4, 5, current-date(), current-time(), 8, 'blue',
               'red', xs:float(3.14), 42, xs:date('1995-04-21'))"/>
  </xsl:variable>

  <xsl:choose>
    <xsl:when test="exists($emptySequence)">
      <xsl:text>The empty sequence does exist!</xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:text>The empty sequence doesn't exist!</xsl:text>
    </xsl:otherwise>
  </xsl:choose>

  <xsl:text>&#xA;  </xsl:text>
  <xsl:value-of select="if (exists($singleton))
                        then 'The singleton sequence does exist!'
                        else 'The singleton sequence doesn''t exist!'"/>

  <xsl:text>&#xA;  </xsl:text>
  <xsl:value-of select="if (exists($longSequence))
                        then 'The long sequence does exist!'
                        else 'The long sequence doesn''t exist!'"/>
</xsl:template>

</xsl:stylesheet>
```

Notice that the first test of `exists( )` does if-then-else logic with the XSLT 1.0 `<choose>` element, while the other tests use the `exists( )` function in XPath 2.0's `if` statement. The results from the stylesheet look like this:

```
Here is a test of the exists() function:

  The empty sequence doesn't exist!
  The singleton sequence does exist!
  The long sequence does exist!
```

false()

Always returns the boolean value `false`. Remember that the strings "true" and "false" don't have any special significance in XSLT; any string with a length greater than zero is true. This function (and the `true( )` function) allow you to generate boolean values when you need them.

Syntax

[1.0] boolean **false()**
[2.0] xs:boolean **false()**

Inputs

None.

Output

The boolean value `false`.

Defined in

XPath section 4.3, "Boolean Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators Section 9.1, "Additional Boolean Constructor Functions."

Example

Here's a brief example that uses the `false( )` function:

```
<?xml version="1.0"?>
<!-- false.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the false() function:&#xA;&#xA;</xsl:text>

    <xsl:text>  false() returned </xsl:text>
    <xsl:value-of select="false()"/>
    <xsl:text>!</xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

When using this stylesheet against any XML document, it generates this less-than exciting result:

```
A test of the false() function:

  false() returned false!
```

floor()

Returns the largest integer that is not greater than the argument.

Syntax

[1.0] number **floor(***number***)**
[2.0] numeric? **floor(***numeric?***)**

Inputs

A number.

[1.0] If the argument is not a number, it is transformed into a number as if it had been processed by the `number( )` function. If the argument cannot be transformed into a number, the `floor( )` function returns the value `NaN` (not a number).

[2.0] In XSLT 2.0, the argument must be one of the four numeric types (`xs:float`, `xs:decimal`, `xs:double`, or `xs:integer`). If it is not, the XSLT processor raises an error. The result of the `floor( )` function will be of the same type as the argument.

In XSLT 2.0, the argument must be a number. If it is not, the XSLT processor raises an error.

Output

The largest integer that is not less than the argument.

[1.0] In XSLT 1.0, `floor( )` returns `NaN` if the argument cannot be converted to a number.

[2.0] In XSLT 2.0, `floor( )` raises an error if the argument cannot be converted to a number.

Defined in

[1.0] XPath section 4.4, "Number Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 6.4, "Functions on Numeric Values."

Example

The following stylesheet shows the results of invoking the `floor( )` function against a variety of values:

```
<?xml version="1.0"?>
<!-- floor.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the floor() function:&#xA;&#xA;</xsl:text>

    <xsl:text>  floor(7.983) = </xsl:text>
    <xsl:value-of select="floor(7.983)"/>

    <xsl:text>&#xA;  floor(-7.893) = </xsl:text>
    <xsl:value-of select="floor(-7.893)"/>

    <xsl:text>&#xA;  floor('blue') = </xsl:text>
    <xsl:value-of version="1.0" select="floor('blue')"/>

  </xsl:template>

</xsl:stylesheet>
```

When we transform the XML document with our stylesheet, here are the results:

```
Tests of the floor() function:

  floor(7.983) = 7
  floor(-7.893) = -8
  floor('blue') = NaN
```

For the last test of the `floor( )` function, we specified `version="1.0"` on the `<xsl:value-of>` element. In XSLT 1.0 mode, we get the result `NaN` (not a number). If we process this `<xsl:value-of>` element in XSLT 2.0 mode, the stylesheet won't run at all. If you change the entire stylesheet to XSLT version 1.0, you'll have to take the `version` attribute off of the `<xsl:value-of>` element.

[2.0] format-date()

Formats an `xs:date` value according to a format string.

Syntax

```
xs:string? format-date(xs:date?, xs:string)
xs:string? format-date(xs:date?, xs:string, $language as xs:string?,
                       $calendar as xs:string?,
                       $country as xs:string?)
```

Inputs

In the first form of this function, an optional `xs:date` value and a formatting string. If the `xs:date` value is omitted, the current date is used. In the second form of this function, you can supply strings that represent the preferred language, calendar, and country codes for the output value.

The following codes are defined for the formatting string. The default values are how the processor interprets codes without any modifiers. In other words, `[Y]` is the same as `[Y1]`. The codes are:

`Y`
: The year. The default value is `[Y1]`, which generates a four-digit year.

`M`
: The month in the year. The default value is `[M1]`, which generates a one- or two-digit numeric month.

`D`
: The day in the month. The default value is `[D1]`, which generates a one- or two-digit numeric day.

`d`
: The day in the year. The default value is `[d1]`, which generates a one-, two-, or three-digit day (if the date is July 26 of a nonleap year, this generates `207`, for example).

F
: The day in the week. The default value is `[Fn]`, which generates the lowercase word representing the day of the week (`wednesday`, for example).

W
: The week in the year. The default value is `[W1]`, which generates the one- or two-digit number of the week in the year.

w
: The week in the month. The default value is `[w1]`, which generates the one-digit number of the week in the month.

Z
: The timezone as an offset from UTC. The default value is `[Z1]`, which generates the hour and minute offset from UTC (`-04:00`, for example).

z
: The timezone as an offset from GMT. The default value is `[z1]`, which generates the offset from GMT (`GMT-4`, for example).

For complete details on the formatting strings for dates, times, and numbers, see Appendix F.

Output

A string representing the `xs:date` formatted according to the format string. If the `xs:date` argument is the empty sequence, the empty sequence is returned.

Defined in

XSLT section 16.5, "Formatting Dates and Times."

Example

This stylesheet illustrates many different ways to format an `xs:date` and its various components:

```
<?xml version="1.0"?>
<!-- format-date.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the format-date() function:&#xA;</xsl:text>

    <xsl:variable name="today" select="current-date()"/>
    <xsl:text>&#xA;  The current date is: </xsl:text>
    <xsl:value-of select="$today"/>
    <xsl:variable name="longFormat"
      select="concat('Today is [FNn], [MNn] [D1], [Y], &#xA;      ',
              'the [D1o] day of the [W1o] week of the year,',
              '&#xA;      the [w1o] week of the month,',
              '&#xA;      in the [zWw] time zone.')"/>
```

```
    <xsl:text>&#xA;&#xA;    The year - format-date</xsl:text>
    <xsl:text>($today, '[Y]'): </xsl:text>
    <xsl:value-of select="format-date($today, '[Y1111]')"/>
    <xsl:text>&#xA;&#xA;    The two-digit year - </xsl:text>
    <xsl:text>format-date($today, '[Y11]'): </xsl:text>
    <xsl:value-of select="format-date($today, '[Y11]')"/>
    <xsl:text>&#xA;&#xA;    The month - format-date(</xsl:text>
    <xsl:text>$today, '[M11]'): </xsl:text>
    <xsl:value-of select="format-date($today, '[M11]')"/>
    <xsl:text>&#xA;&#xA;    The month in German words - </xsl:text>
    <xsl:text>format-date($today, '[MWw]', 'de', (), ()): </xsl:text>
    <xsl:text>&#xA;      </xsl:text>
    <xsl:value-of
      select="format-date($today, '[MWw]', 'de', (), ())"/>
    <xsl:text>&#xA;&#xA;    The day in words - format-date(</xsl:text>
    <xsl:text>$today, '[DWw]'): &#xA;      </xsl:text>
    <xsl:value-of select="format-date($today, '[DWw]')"/>
    <xsl:text>&#xA;&#xA;    The day in German words - </xsl:text>
    <xsl:text>format-date($today, '[Dw]', 'de', (), ()): </xsl:text>
    <xsl:text>&#xA;      </xsl:text>
    <xsl:value-of select="format-date($today, '[Dw]', 'de', (), ())"/>
    <xsl:text>&#xA;&#xA;    Using format-date($today, '[dwo]'):</xsl:text>
    <xsl:text>&#xA;      It's the </xsl:text>
    <xsl:value-of select="format-date($today, '[dwo]')"/>
    <xsl:text> day of the year.</xsl:text>
    <xsl:text>&#xA;&#xA;    Using format-date($today, '[d1o]'):</xsl:text>
    <xsl:text>&#xA;      It's the </xsl:text>
    <xsl:value-of select="format-date($today, '[d1o]')"/>
    <xsl:text> day of the year.</xsl:text>

    <xsl:text>&#xA;&#xA;  The grand finale:</xsl:text>
    <xsl:text>&#xA;    </xsl:text>
    <xsl:value-of select="format-date($today, $longFormat)"/>
  </xsl:template>

</xsl:stylesheet>
```

The results look like this:

```
Tests of the format-date() function:

  The current date is: 2006-07-26-04:00

    The year - format-date($today, '[Y]'): 2006

    The two-digit year - format-date($today, '[Y11]'): 06

    The month - format-date($today, '[M11]'): 07

    The month in German words - format-date($today, '[MWw]', 'de', (), ()):
      Sieben

    The day in words - format-date($today, '[DWw]'):
      Twenty Six
```

```
    The day in German words - format-date($today, '[Dw]', 'de', (), ()):
      sechsundzwanzig

    Using format-date($today, '[dwo]):
      It's the two hundred and seventh day of the year.

    Using format-date($today, '[d1o]):
      It's the 207th day of the year.

  The grand finale:
    Today is Wednesday, July 26, 2006,
      the 26th day of the 30th week of the year,
      the 4th week of the month,
      in the GMT-4 time zone.
```

Notice that we used the language code `de` for two of the examples. With the different combinations of modifiers, we printed out the components of a date in numbers and words, both in cardinal and ordinal formats. Also notice that we stored the formatting string in a variable so we could use it whenever we want.

[2.0] format-dateTime()

Formats an `xs:dateTime` value according to a format string.

Syntax

```
xs:string? format-dateTime(xs:dateTime?, $picture as xs:string)
xs:string? format-dateTime(xs:dateTime?, $picture as xs:string,
                           $language as xs:string?,
                           $calendar as xs:string?,
                           $country as xs:string?)
```

Inputs

In the first form of this function, an optional `xs:dateTime` value and a formatting string. If the `xs:dateTime` value is omitted, the current date and time is used. In the second form of this function, you can supply strings that represent the preferred language, calendar, and country codes for the output value.

The following codes are defined for the formatting string:

`Y`
: The year. The default value is `[Y1]`, which generates a four-digit year.

`M`
: The month in the year. The default value is `[M1]`, which generates a one- or two-digit numeric month.

`D`
: The day in the month. The default value is `[D1]`, which generates a one- or two-digit numeric day.

`d`

The day in the year. The default value is `[d1]`, which generates a one-, two-, or three-digit day (if the date is July 26 of a nonleap year, this generates `207`, for example).

`F`

The day in the week. The default value is `[Fn]`, which generates the lowercase word representing the day of the week (`wednesday`, for example).

`W`

The week in the year. The default value is `[W1]`, which generates the one- or two-digit number of the week in the year.

`w`

The week in the month. The default value is `[w1]`, which generates the one-digit number of the week in the month.

`H`

The hour in the day, in a 24-hour clock. The default value is `[H1]`, which generates the one- or two-digit number of the hour in the day. Note that this value will never be greater than `23`.

`h`

The hour in the day, in a 12-hour clock. The default value is `[h1]`, which generates the one- or two-digit number of the hour in the day.

`P`

The AM/PM marker. The default value is `[Pn]`, which generates the AM/PM marker in lowercase (`a.m.` or `p.m.`).

`m`

The minute in the hour. The default value is `[m01]`, which generates the two-digit minute in the hour.

`s`

The second in the minute. The default value is `[s01]`, which generates the two-digit second in the minute.

`f`

The fractional seconds. The default value is `[f1]`, which generates the fractional seconds to three decimal places.

`Z`

The timezone as an offset from UTC. The default value is `[Z1]`, which generates the hour and minute offset from UTC (`-04:00`, for example).

`z`

The timezone as an offset from GMT. The default value is `[z1]`, which generates the offset from GMT (`GMT-4`, for example).

For complete details on the formatting strings for dates, times, and numbers, see Appendix F.

Output

A string representing the `xs:dateTime` formatted according to the format string. If the `xs:dateTime` argument is the empty sequence, the empty sequence is returned.

Defined in

XSLT section 16.5, "Formatting Dates and Times."

Example

This stylesheet illustrates many different ways to format an `xs:dateTime` and its various components:

```
<?xml version="1.0"?>
<!-- format-dateTime.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the format-dateTime() function:&#xA;</xsl:text>

    <xsl:variable name="today" select="current-dateTime()"/>
    <xsl:text>&#xA;  The current date and time is: </xsl:text>
    <xsl:value-of select="$today"/>
    <xsl:variable name="longFormat"
      select="concat('It''s [h1]:[m11] [P] on ',
              '[FNn], [MNn] [D1], [Y], &#xA;      ',
              'the [D1o] day of the [W1o] week of the year,',
              '&#xA;      the [w1o] week of the month,',
              '&#xA;      in the [zWw] time zone.')"/>

    <xsl:text>&#xA;&#xA;    The year - format-dateTime</xsl:text>
    <xsl:text>($today, '[Y]'): </xsl:text>
    <xsl:value-of select="format-dateTime($today, '[Y]')"/>
    <xsl:text>&#xA;&#xA;    The hour (12-hour clock) - </xsl:text>
    <xsl:text>format-dateTime($today, '[h] [PN]'): </xsl:text>
    <xsl:value-of select="format-dateTime($today, '[h] [PN]')"/>
    <xsl:text>&#xA;&#xA;    The minutes - format-dateTime(</xsl:text>
    <xsl:text>$today, '[m11]'): </xsl:text>
    <xsl:value-of select="format-dateTime($today, '[m11]')"/>
    <xsl:text>&#xA;&#xA;    The seconds - format-dateTime(</xsl:text>
    <xsl:text>$today, '[s11].[f1111]): </xsl:text>
    <xsl:value-of select="format-dateTime($today, '[s11].[f1111]')"/>
    <xsl:text>&#xA;&#xA;    The complete time - format-dateTime(</xsl:text>
    <xsl:text>$today, '[h]:[m11]:[s11] [P]'): &#xA;      </xsl:text>
    <xsl:value-of
      select="format-dateTime($today, '[h]:[m11]:[s11] [P]')"/>
    <xsl:text>&#xA;&#xA;    The day in words - format-dateTime(</xsl:text>
    <xsl:text>$today, '[FNn]'): &#xA;      </xsl:text>
    <xsl:value-of select="format-dateTime($today, '[FNn]')"/>

    <xsl:text>&#xA;&#xA;  The grand finale:</xsl:text>
```

```
    <xsl:text>&#xA;    </xsl:text>
    <xsl:value-of select="format-dateTime($today, $longFormat)"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Tests of the format-dateTime() function:

  The current date and time is: 2006-07-26T13:58:25.790-04:00

    The year - format-dateTime($today, '[Y]'): 2006

    The hour (12-hour clock) - format-dateTime($today, '[h] [PN]'): 1 P.M.

    The minutes - format-dateTime($today, '[m11]'): 58

    The seconds - format-dateTime($today, '[s11].[f1111]): 25.7900

    The complete time - format-dateTime($today, '[h]:[m11]:[s11] [P]'):
      1:58:25 p.m.

    The day in words - format-dateTime($today, '[FNn]'):
      Wednesday

  The grand finale:
    It's 1:58 p.m. on Wednesday, July 26, 2006,
      the 26th day of the 30th week of the year,
      the 4th week of the month,
      in the GMT-4 time zone.
```

format-number()

Takes a number and formats it as a string.

Syntax

```
[1.0] string format-number($value as number, $picture as string,
                           $formatName as string?)
[2.0] xs:string format-number($value as numeric, $picture as xs:string,
                              $formatName as xs:string?)
```

Inputs

The number to be formatted and the format pattern string are required. The third argument is the optional name of a decimal format; if the third argument is not supplied, the default decimal format is used.

[2.0] The numeric argument must be one of the four numeric types (`xs:float`, `xs:decimal`, `xs:double`, or `xs:integer`). If it is not a numeric type or cannot be converted to one, the XSLT processor raises an error. The XSLT processor also raises an error if the named decimal format does not exist.

Output

The number, formatted according to the rules supplied by the other arguments. The special characters used in the second argument are:

`#`
: Represents a digit. Trailing or leading zeroes are not displayed. Formatting the number `4.0` with the string `#.##` returns the string `4`.

`0`
: Represents a digit. Unlike the `#` character, the `0` always displays a zero. Formatting the number `4.1` with the string `#.00` returns the string `4.10`.

`.`
: Represents the decimal point.

`-`
: Represents the minus sign.

`,`
: Is the grouping separator.

`;`
: Separates the positive-number pattern from the negative-number pattern.

`%`
: Indicates that a number should be displayed as a percentage. The value will be multiplied by 100, then displayed as a percentage. Formatting the number `.76` with the string `##%` returns the string `76%`.

`\u2030`
: Is the Unicode character for the per-thousand (per-mille) sign. The value will be multiplied by 1000, then displayed as a per mille. Formatting the number `.768` with the string `###\u2030` returns the string `768‰`.

For complete details on the formatting strings for dates, times and numbers, see Appendix F.

The third argument, if given, must be the name of an `<xsl:decimal-format>` element. The `<xsl:decimal-format>` element lets you define the character that should be used for the decimal point and the grouping separator, the string used to represent infinity, and other formatting options. See the discussion of the `<xsl:decimal-format>` element in Appendix A for more information.

Defined in

[1.0] XSLT section 12.3, "Number Formatting."

[2.0] XSLT section 16.4, "Number Formatting."

Example

The following stylesheet uses the `format-number( )` function in various ways:

```
<?xml version="1.0"?>
<!-- format-number.xsl -->
```

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:decimal-format name="f1"
    decimal-separator=":"
    grouping-separator="/"/>

  <xsl:decimal-format name="f2"
    infinity="Really, really big"
    NaN="[not a number]"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the format-number() </xsl:text>
    <xsl:text>function:&#xA;&#xA;</xsl:text>

    <xsl:text>  1. format-number(528.3, '#.#;-#.#')=</xsl:text>
    <xsl:value-of
      select="format-number(528.3, '#.#;-#.#')"/>
    <xsl:text>&#xA;  2. format-number(528.3, </xsl:text>
    <xsl:text>'0,000.00;-0,000.00')=</xsl:text>
    <xsl:value-of
      select="format-number(528.3, '0,000.00;-0,000.00')"/>
    <xsl:text>&#xA;  3. format-number(-23528.3, </xsl:text>
    <xsl:text>'$#,###.00;($#,###.00)')=</xsl:text>
    <xsl:value-of
      select="format-number(-23528.3, '$#,###.00;($#,###.00)')"/>
    <xsl:text>&#xA;  4. format-number(1528.3, </xsl:text>
    <xsl:text>'#/###:00', 'f1')=</xsl:text>
    <xsl:value-of
      select="format-number(1528.3, '#/###:00;-#/###:00', 'f1')"/>
    <xsl:text>&#xA;  5. format-number(1 div 0, </xsl:text>
    <xsl:text>'###,###.00', 'f2')=</xsl:text>
    <xsl:value-of
      select="format-number(1 div 0, '###,###.00', 'f2')"/>
    <xsl:text>&#xA;  6. format-number(blue div orange, </xsl:text>
    <xsl:text>'#.##', 'f2')=</xsl:text>
    <xsl:value-of
      select="format-number(blue div orange, '#.##', 'f2')"/>
    <xsl:text>&#xA;&#xA;  </xsl:text>

    <xsl:value-of select="report/title"/>
    <xsl:text> - </xsl:text>
    <xsl:value-of select="report/@month"/>
    <xsl:text>/</xsl:text>
    <xsl:value-of select="report/@year"/>
    <xsl:text>&#xA;&#xA;</xsl:text>

    <xsl:variable name="totalSales"
      select="sum(/report/brand/units)"/>

    <xsl:for-each select="report/brand">
      <xsl:text>    </xsl:text>
      <xsl:value-of select="name"/>
```

```
      <xsl:text> - </xsl:text>
      <xsl:value-of select="format-number(units, '##,###')"/>
      <xsl:text> bars sold, </xsl:text>
      <xsl:value-of
        select="format-number(units div $totalSales, '##%')"/>
      <xsl:text> of all sales.&#xA;</xsl:text>
    </xsl:for-each>

    <xsl:text>&#xA;  Total sales: </xsl:text>
    <xsl:value-of
      select="format-number(sum(/report/brand/units), '##,###')"/>
  </xsl:template>

</xsl:stylesheet>
```

We'll use this XML document with our stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  ...
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

When we run this stylesheet, here are the results:

```
Tests of the format-number() function:

  1. format-number(528.3, '#.#;-#.#')=528.3
  2. format-number(528.3, '0,000.00;-0,000.00')=0,528.30
  3. format-number(-23528.3, '$#,###.00;($#,###.00)')=($23,528.30)
  4. format-number(1528.3, '#/###:00', 'f1')=1/528:30
  5. format-number(1 div 0, '###,###.00', 'f2')=Really, really big
  6. format-number(blue div orange, '#.##', 'f2')=[not a number]

  Chocolate bar sales - 8/2006

    Lindt - 27,408 bars sold, 30% of all sales.
    Callebaut - 8,203 bars sold, 9% of all sales.
    Valrhona - 22,101 bars sold, 24% of all sales.
    Perugina - 14,336 bars sold, 16% of all sales.
    Ghirardelli - 19,268 bars sold, 21% of all sales.

  Total sales: 91,316
```

The first three examples use `format-number( )` to control the decimal places, commas, and the format of negative numbers. The fourth example uses the `<decimal-format>` named `f1`, which uses a slash (/) as a grouping separator and a colon (:) as the decimal point.

Examples 5 and 6 cause problems in XSLT 2.0. Example 5 doesn't work because division by zero is a fatal error for `xs:integer` values in XSLT 2.0. Example 6 doesn't work because the `div` operator requires numeric operands. If you want XSLT 1.0 processing in an XSLT 2.0 stylesheet, you can add the `version="1.0"` attribute to any `<xsl:value-of>` element that needs it:

```
<xsl:value-of version="1.0"
  select="format-number(1 div 0, '###,###.00', 'f2')"/>
```

The last set of examples extracts data from our document of chocolate bar sales, formatting the total sales of each bar. We also generate the percentage of total sales for each brand and print the total sales for all brands.

[2.0] format-time()

Given an `xs:time` value and a format string, returns the formatted time value.

Syntax

```
xs:string format-time(xs:time?, $picture as xs:string)
xs:string format-time(xs:time?, $picture as xs:string,
                      $language as xs:string?,
                      $calendar as xs:string?,
                      $country as xs:string?)
```

Inputs

In the first form of this function, an optional `xs:time` value and a formatting string. If the `xs:time` value is omitted, the current time is used. In the second form of this function, you can supply strings that represent the preferred language, calendar, and country codes for the output value.

The following codes are defined for the formatting string:

`H`
: The hour in the day, in a 24-hour clock. The default value is `[H1]`, which generates the one- or two-digit number of the hour in the day. Note that this value will never be greater than `23`.

`h`
: The hour in the day, in a 12-hour clock. The default value is `[h1]`, which generates the one- or two-digit number of the hour in the day.

`P`
: The AM/PM marker. The default value is `[Pn]`, which generates the AM/PM marker in lowercase (`a.m.` or `p.m.`).

`m`
: The minute in the hour. The default value is `[m01]`, which generates the two-digit minute in the hour.

`s`

The second in the minute. The default value is `[s01]`, which generates the two-digit second in the minute.

`f`

The fractional seconds. The default value is `[f1]`, which generates the fractional seconds to three decimal places.

`Z`

The timezone as an offset from UTC. The default value is `[Z1]`, which generates the hour and minute offset from UTC (`-04:00`, for example).

`z`

The timezone as an offset from GMT. The default value is `[z1]`, which generates the offset from GMT (`GMT-4`, for example).

For complete details on the formatting strings for dates, times, and numbers, see Appendix F.

Output

A string representing the `xs:time` formatted according to the format string. If the `xs:time` argument is the empty sequence, the empty sequence is returned.

Defined in

XSLT section 16.5, "Formatting Dates and Times."

Example

Here's a stylesheet that uses `format-time( )` in several different ways:

```
<?xml version="1.0"?>
<!-- format-time.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the format-time() function:&#xA;</xsl:text>

    <xsl:variable name="now" select="current-time()"/>
    <xsl:text>&#xA;  The current date and time is: </xsl:text>
    <xsl:value-of select="$now"/>
    <xsl:variable name="longFormat"
      select="concat('It''s [h1]:[m11]:[s11] [P] ',
              'in the [zWw] time zone.')"/>

    <xsl:text>&#xA;&#xA;    The hour (24-hour clock) - </xsl:text>
    <xsl:text>format-time($now, '[H]'): </xsl:text>
    <xsl:value-of select="format-time($now, '[H]')"/>
    <xsl:text>&#xA;&#xA;    The minutes - format-time(</xsl:text>
    <xsl:text>$now, '[m11]'): </xsl:text>
    <xsl:value-of select="format-time($now, '[m11]')"/>
```

```
    <xsl:text>&#xA;&#xA;    Using format-time(</xsl:text>
    <xsl:text>$now, '[mWwo]'): &#xA;      It's the </xsl:text>
    <xsl:value-of select="format-time($now, '[mWwo]')"/>
    <xsl:text> minute of the hour.</xsl:text>
    <xsl:text>&#xA;&#xA;    The time - format-time(</xsl:text>
    <xsl:text>$now, '[H]:[m11]'): &#xA;      </xsl:text>
    <xsl:value-of
      select="format-time($now, '[H]:[m11]')"/>

    <xsl:text>&#xA;&#xA;  The grand finale:</xsl:text>
    <xsl:text>&#xA;    </xsl:text>
    <xsl:value-of select="format-time($now, $longFormat)"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Tests of the format-time() function:

  The current date and time is: 14:10:00.028-04:00

    The hour (24-hour clock) - format-time($now, '[H]'): 14

    The minutes - format-time($now, '[m11]'): 10

    Using format-time($now, '[mWwo]'):
      It's the Tenth minute of the hour.

    The time - format-time($now, '[H]:[m11]'):
      14:10

  The grand finale:
    It's 2:10:00 p.m. in the GMT-4 time zone.
```

function-available()

Determines if a given function is available to the XSLT processor. This function allows you to design stylesheets that react gracefully if a particular extension function is not available.

Syntax

```
[1.0] boolean function-available($functionName as string)
[2.0] xs:boolean function-available($functionName as xs:string,
                                    $arity as xs:integer?)
```

Inputs

The function's name. The name is usually qualified with a namespace; if the namespace of the function name is nonnull, the function is an extension function. Otherwise, the function is assumed to be one of the functions defined in the XSLT or XPath specifications.

[2.0] XSLT 2.0's version of this function also has an *arity* argument. This lets you specify how many parameters a function accepts. For example, `function-available('ora:greatFunction',`

2) might return true, while function-available('ora:greatFunction', 3) might return false. If you try to call function-available('me:my-function', 2) with an XSLT 1.0 processor, you'll get a runtime error.

Output

The boolean value true if the function is available; false otherwise.

Defined in

[1.0] XSLT section 15, "Fallback."

[2.0] XSLT section 18.1, "Extension Functions."

Example

We'll use the following XML document to test the function-available( ) function:

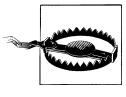
```
<?xml version="1.0"?>
<!-- favorites.xml -->
<list>
  <title>A few of my favorite albums</title>
  <listitem>A Love Supreme</listitem>
  <listitem>Beat Crazy</listitem>
  <listitem>Here Come the Warm Jets</listitem>
  <listitem>Kind of Blue</listitem>
  <listitem>London Calling</listitem>
  <listitem>Remain in Light</listitem>
  <listitem>The Joshua Tree</listitem>
  <listitem>The Indestructible Beat of Soweto</listitem>
</list>
```

Here's our stylesheet:

```
<?xml version="1.0"?>
<!-- function-available.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:jpeg="class:JPEGWriter"
  extension-element-prefixes="jpeg">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of function-avilable():&#xA;&#xA;</xsl:text>
    <xsl:for-each select="list/listitem">
      <xsl:choose>
        <xsl:when test="function-available('jpeg:buildJPEGFile')">
          <xsl:value-of
            select="jpeg:buildJPEGFile(., 'titlebar.jpg',
            concat('album', position(), '.jpg'),
            'Verdana Bold Italic', 20, 12, 20, '000000')"/>
          <xsl:text>See the file </xsl:text>
          <xsl:value-of select="concat('album', position(), '.jpg')"/>
          <xsl:text> to see the title of album #</xsl:text>
          <xsl:value-of select="position()"/>
          <xsl:text>&#xA;</xsl:text>
```

Figure C-3. Generated graphic for the eighth <listitem> element

```
          </xsl:when>
          <xsl:otherwise>
            <xsl:value-of select="position()"/>
            <xsl:text>. </xsl:text>
            <xsl:value-of select="."/>
            <xsl:text>&#xA;</xsl:text>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:for-each>
    </xsl:template>

</xsl:stylesheet>
```



Be aware that the mechanism for defining, referencing, and building extension functions varies from one processor to another. The example here uses Xalan-J's mechanism. We create a new namespace (`jpeg`) and define its URI as `class:JPEGWriter`. When we ask whether a function in the `jpeg` namespace is available, Xalan-J attempts to load a Java class named `JPEGWriter`. If that class can't be found, `function-available( )` returns `false`. If the class is found, Xalan-J looks in the class and returns `true` if that Java class contains a method with the requested name (`buildJPEGFile`, in this example).

Also be aware that `function-available( )` is not required to check the signature of the method. In our example, if Xalan-J finds the class and a method with the name we requested, it returns `true`. That doesn't mean that Xalan-J checked the number and type of arguments that we're passing to the function to see whether they're correct. If they aren't correct, we'll get a runtime error.

In our stylesheet, if the `buildJPEGFile( )` function is available, we'll invoke it to create JPEG files for the titles of all our favorite albums. If the function is not available, we'll simply write those titles to the output stream. Here are the results we get when the `buildJPEGFile( )` function is available:

```
See the file album1.jpg to see the title of album #1
See the file album2.jpg to see the title of album #2
See the file album3.jpg to see the title of album #3
See the file album4.jpg to see the title of album #4
See the file album5.jpg to see the title of album #5
See the file album6.jpg to see the title of album #6
See the file album7.jpg to see the title of album #7
See the file album8.jpg to see the title of album #8
```

All album titles (the text of the `<listitem>` elements) are converted to JPEG graphics. In this example, the file *album8.jpg* looks like Figure C-3.

If the XSLT processor can't load the extension function (maybe we're using Saxon, which doesn't load extension classes this way; maybe the *.class* file isn't on the classpath; maybe we misspelled the function name, etc.), we get these results instead:

```
1. A Love Supreme
2. Beat Crazy
3. Here Come the Warm Jets
4. Kind of Blue
5. London Calling
6. Remain in Light
7. The Joshua Tree
8. The Indestructible Beat of Soweto
```

generate-id()

Generates a unique ID (an XML name) for a given node. If no node-set is given, `generate-id( )` generates an ID for the context node.

Syntax

```
[1.0] string generate-id(node-set?)
[2.0] xs:string generate-id()
[2.0] xs:string generate-id(node()?)
```

Inputs

[1.0] An optional node-set. If no node-set is given, this function generates an ID for the context node. If the node-set is empty, `generate-id( )` returns an empty string.

[2.0] An optional node. If no node is given, this function generates an ID for the context node. If the argument is the empty sequence, the result is a zero-length string.

In XSLT 1.0, passing a node-set to the `generate-id( )` function generated a unique ID for the first item in the node-set; all nodes after the first were ignored. *In XSLT 2.0, it is an error to pass a sequence with more than one node to* `generate-id( )`.

Output

A unique ID, or an empty string if an empty node-set is given. Several things about the `generate-id( )` function are important to know:

- For a given transformation, every time you invoke `generate-id( )` against a given node, the XSLT processor must return the same ID. The ID can't change while you're doing a transformation. If you ask the XSLT processor to transform your document with this stylesheet tomorrow, there's no guarantee that `generate-id( )` will generate the same ID the second time around. All of tomorrow's calls (during one transformation) to `generate-id( )` will generate the same ID, but that ID might not be the one generated today.
- The `generate-id( )` function is not required to check whether its generated ID duplicates an ID that's already in the document. In other words, if an element in your document has an attribute of type `ID` with the value `sdk3829a`, there's a remote

possibility that an ID returned by `generate-id( )` would have the value `sdk3829a`. It's not likely, but it could happen.

- If you invoke `generate-id( )` against two different nodes, the two generated IDs must be different.
- *[1.0]* Given a node-set, `generate-id( )` returns an ID for the node in the node-set that occurs first in document order.

  [2.0] Given a sequence, an XSLT 2.0 processor raises an error if that sequence contains more than one item.
- If the node-set you pass to the function is empty (you invoke `generate-id(fleeber)`, but there are no `<fleeber>` elements in the current context), `generate-id( )` returns an empty string.

Defined in

[1.0] XSLT section 12.4, "Miscellaneous Additional Functions."

[2.0] XSLT section 16.6, "Miscellaneous Additional Functions."

Example

We'll use our report of chocolate bar sales as the input to our stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

Our stylesheet generates a new ID for each node in our source document. Just to make sure the XSLT processor generates the same node each time, we generate a new ID for each node again. The two generated IDs should match for each node:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the generate-id() </xsl:text>
    <xsl:text>function:&#xA;&#xA;</xsl:text>

    <xsl:text>  We'll generate IDs for every node </xsl:text>
    <xsl:text>in the XML source:&#xA;&#xA;</xsl:text>
    <xsl:for-each select="//*">
      <xsl:text>    Node name: &lt;</xsl:text>
      <xsl:value-of select="name()"/>
      <xsl:text>&gt; - generated id: </xsl:text>
      <xsl:value-of select="generate-id()"/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>

    <xsl:text>&#xA;  Now we'll try it again...&#xA;&#xA;</xsl:text>
    <xsl:for-each select="//*">
      <xsl:text>    Node name: &lt;</xsl:text>
      <xsl:value-of select="name()"/>
      <xsl:text>&gt; - generated id: </xsl:text>
      <xsl:value-of select="generate-id()"/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Our stylesheet generates these results:

```
A test of the generate-id() function:

  We'll generate IDs for every node in the XML source:

    Node name: <report> - generated id: d1e2
    Node name: <title> - generated id: d1e4
    Node name: <brand> - generated id: d1e7
    Node name: <name> - generated id: d1e9
    Node name: <units> - generated id: d1e12
    Node name: <brand> - generated id: d1e16
    Node name: <name> - generated id: d1e18
    Node name: <units> - generated id: d1e21
    Node name: <brand> - generated id: d1e25
    Node name: <name> - generated id: d1e27
    Node name: <units> - generated id: d1e30
    Node name: <brand> - generated id: d1e34
    Node name: <name> - generated id: d1e36
    Node name: <units> - generated id: d1e39
    Node name: <brand> - generated id: d1e44
    Node name: <name> - generated id: d1e46
    Node name: <units> - generated id: d1e49
```

```
Now we'll try it again...

  Node name: <report> - generated id: d1e2
  Node name: <title> - generated id: d1e4
  Node name: <brand> - generated id: d1e7
  Node name: <name> - generated id: d1e9
  Node name: <units> - generated id: d1e12
  Node name: <brand> - generated id: d1e16
  Node name: <name> - generated id: d1e18
  Node name: <units> - generated id: d1e21
  Node name: <brand> - generated id: d1e25
  Node name: <name> - generated id: d1e27
  Node name: <units> - generated id: d1e30
  Node name: <brand> - generated id: d1e34
  Node name: <name> - generated id: d1e36
  Node name: <units> - generated id: d1e39
  Node name: <brand> - generated id: d1e44
  Node name: <name> - generated id: d1e46
  Node name: <units> - generated id: d1e49
```

The IDs generated each time are the same. Be aware that other processors generate IDs differently. The first ID generated by Xalan-J is `N10002`, while the first ID generated by AltovaXML is `idreport220739936`.

[2.0] hours-from-dateTime()

Given an `xs:dateTime` value, returns its `hours` value.

Syntax

```
xs:integer? hours-from-dateTime(xs:dateTime?)
```

Inputs

An `xs:dateTime` value.

Output

An `xs:integer` representing the hours component of the given `xs:dateTime` value. If the input argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `hours-from-dateTime( )` function:

```
<?xml version="1.0"?>
<!-- hours-from-datetime.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xsl:output method="text"/>

    <xsl:template match="/">
      <xsl:text>&#xA;Extracting the hour from an xs:dateTime:</xsl:text>
      <xsl:variable name="currentDateTime" as="xs:dateTime"
        select="current-dateTime()"/>
      <xsl:text>&#xA;&#xA;  The current date and time is: </xsl:text>
      <xsl:value-of select="$currentDateTime"/>

      <xsl:text>&#xA;&#xA;  The current hour is: </xsl:text>
      <xsl:value-of select="hours-from-dateTime($currentDateTime)"/>
      <xsl:text>&#xA;    Also known as </xsl:text>
      <xsl:value-of select="format-dateTime($currentDateTime, '[h] [Pn]')"/>
      <xsl:text>&#xA;    or </xsl:text>
      <xsl:value-of select="format-dateTime($currentDateTime, '[H] [ZN]')"/>
      <xsl:text>&#xA;    or </xsl:text>
      <xsl:value-of select="format-dateTime($currentDateTime, '[H] [z]')"/>
    </xsl:template>

</xsl:stylesheet>
```

The stylesheet creates these results:

```
Extracting the hour from an xs:dateTime:

  The current date and time is: 2006-11-16T04:17:15.778-05:00

  The current hour is: 4
    Also known as 4 a.m.
    or 4 EST
    or 4 GMT-5
```

Notice that some of the results were generated by the `hours-from-dateTime( )` function, while others were generated by using `format-dateTime( )` with a format string that selected only the hours component of the `xs:dateTime` value.

See Also

The definitions of the [2.0] `day-from-dateTime( )`, [2.0] `format-dateTime( )`, [2.0] `minutes-from-dateTime( )`, [2.0] `month-from-dateTime( )`, [2.0] `seconds-from-dateTime( )`, [2.0] `time zone-from-dateTime( )`, and [2.0] `year-from-dateTime( )` functions.

[2.0] hours-from-duration()

Given an `xs:duration` value, returns the number of hours in that duration.

Syntax

```
xs:integer? hours-from-duration(xs:duration?)
```

Inputs

An `xs:duration`.

Output

An `xs:integer` representing the hours component of the given `xs:duration`. Be aware that for an `xs:yearMonthDuration`, this function always returns `0` because there is no hours component of an `xs:yearMonthDuration`. Also, if the input argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `hours-from-duration( )` function with all three types of durations:

```
<?xml version="1.0"?>
<!-- hours-from-duration.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the hours component from durations:</xsl:text>

    <xsl:variable name="sampleDuration" as="xs:duration"
      select="xs:duration('P3Y8M2DT4H23M12.2S')"/>
    <xsl:variable name="sampleYearMonthDuration" as="xs:yearMonthDuration"
      select="xs:yearMonthDuration('P3Y8M')"/>
    <xsl:variable name="sampleDayTimeDuration" as="xs:dayTimeDuration"
      select="xs:dayTimeDuration('P2DT4H23M12.2S')"/>

    <xsl:text>&#xA;&#xA;  A sample xs:duration: </xsl:text>
    <xsl:value-of select="$sampleDuration"/>
    <xsl:text>&#xA;    The hours component of this duration is </xsl:text>
    <xsl:value-of select="hours-from-duration($sampleDuration)"/>
    <xsl:text>.</xsl:text>

    <xsl:text>&#xA;&#xA;  A sample xs:yearMonthDuration: </xsl:text>
    <xsl:value-of select="$sampleYearMonthDuration"/>
    <xsl:text>&#xA;    The hours component of this duration is </xsl:text>
    <xsl:value-of select="hours-from-duration($sampleYearMonthDuration)"/>
    <xsl:text>.</xsl:text>

    <xsl:text>&#xA;&#xA;  A sample xs:dayTimeDuration: </xsl:text>
    <xsl:value-of select="$sampleDayTimeDuration"/>
    <xsl:text>&#xA;    The hours component of this duration is </xsl:text>
    <xsl:value-of select="hours-from-duration($sampleDayTimeDuration)"/>
```

```
    <xsl:text>.</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates these results:

```
Extracting the hours component from durations:

  A sample xs:duration: P3Y8M2DT4H23M12.2S
    The hours component of this duration is 4.

  A sample xs:yearMonthDuration: P3Y8M
    The hours component of this duration is 0.

  A sample xs:dayTimeDuration: P2DT4H23M12.2S
    The hours component of this duration is 4.
```

Notice that calling `hours-from-duration( )` against `xs:yearMonthDuration` returns `0`.

See Also

The definitions of the [2.0] `days-from-duration( )`, [2.0] `minutes-from-duration( )`, [2.0] `months-from-duration( )`, [2.0] `seconds-from-duration( )`, and [2.0] `years-from-duration( )` functions.

[2.0] hours-from-time()

Given an `xs:time` value, returns its hours component.

Syntax

```
xs:integer? hours-from-time(xs:time?)
```

Input

An `xs:time` value.

Output

An `xs:integer` representing the hours component of the given `xs:time` value. If the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet uses `hours-from-time( )` against the current time:

```
<?xml version="1.0"?>
<!-- hours-from-time.xsl -->
<xsl:stylesheet version="2.0"
```

```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xsl:output method="text"/>

    <xsl:template match="/">
      <xsl:text>&#xA;Extracting the hours component from an xs:time:</xsl:text>
      <xsl:variable name="currentTime" as="xs:time" select="current-time()"/>
      <xsl:text>&#xA;&#xA;  The current time is: </xsl:text>
      <xsl:value-of select="$currentTime"/>

      <xsl:text>&#xA;&#xA;  The current hour is: </xsl:text>
      <xsl:value-of select="hours-from-time($currentTime)"/>
      <xsl:text>&#xA;    Also known as </xsl:text>
      <xsl:value-of select="format-time($currentTime, '[h] [Pn]')"/>
      <xsl:text>, </xsl:text>
      <xsl:value-of select="format-time($currentTime, '[H] [ZN]')"/>
      <xsl:text>, or </xsl:text>
      <xsl:value-of select="format-time($currentTime, '[H] [z]')"/>
    </xsl:template>

</xsl:stylesheet>
```

The stylesheet creates these results:

```
Extracting the hours component from an xs:time:

  The current time is: 17:02:52.515-05:00

  The current hour is: 17
    Also known as 5 p.m., 17 EST, or 17 GMT-5
```

Notice that the first result was generated by the `hours-from-time( )` function, while the other results were generated by the `format-time( )` function with a format string that selected the hours component from the `xs:time` value.

See Also

The definitions of the [2.0] `format-time( )`, [2.0] `minutes-from-time( )`, [2.0] `seconds-from-time( )`, and [2.0] `timezone-from-time( )` functions.

id()

Returns the node in the source tree whose ID attribute matches the value(s) passed in as input.

Syntax

```
[1.0] node-set id(object)
[2.0] element()* id(xs:string*)
[2.0] element()* id(xs:string*, node())
```

Inputs

[1.0] An object. If the input object is a node-set, the result is a node-set that contains the result of applying the `id( )` function to the string value of each node in the argument node-set. Usually, the argument is some other node type, which is (or is converted to) a string. That string is then used as the search value while all attributes of type `ID` are searched.

[2.0] An `xs:string` and an optional node. If the optional node argument is present, the processor looks for those values in the document that contains that node.

In both XSLT 1.0 and 2.0, if the string value contains multiple values separated by spaces, each space-separated value is used as a search argument for the `id( )` function. Our examples here use both simple strings and strings with multiple values.

Remember that a limitation of the XML `ID` datatype is that a single set of names across all attributes is declared to be of type `ID`. The XSLT `key( )` function and the associated `<xsl:key>` element address this and other limitations; see the description of the `key( )` function earlier in this appendix and the description of the `<xsl:key>` element in Appendix A for more information.

Output

[1.0] A node-set containing all nodes whose attributes of type `ID` match the string values of the input node-set.

[2.0] A sequence containing all the nodes that have an `ID` value equal to one of the values specified in the search string.

In a validated XML document, only one node will match each search value; the value of an `ID` attribute must be unique across all `ID` attributes in the document. Given that an XSLT processor is not required to validate an XML document, it is possible that the node-set or sequence will have more than one match for each search value.

Defined in

[1.0] XPath section 4.1, "Node Set Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 15.5, "Functions and Operators that Generate Sequences."

Example

For our example, we'll use a list of components and parts with an embedded DTD. We'll list each component, followed by the name of each part used by that component. The `id( )` function lets us retrieve the `<part>` element with the specific `ID`; from there we can get the name of the part.

This is the exact opposite of the example for the XSLT 2.0 `idref( )` function. In that example, we take each `<part>` and list all of the components that reference it.

Here's our XML document:

```
<?xml version="1.0"?>
<!-- parts-list1.xml -->
<!DOCTYPE parts-list [
<!ELEMENT parts-list       (component+, part+)>

<!ELEMENT component        (name, partref+)>
<!ATTLIST component        component-id ID #REQUIRED>

<!ELEMENT name             (#PCDATA)>

<!ELEMENT partref          EMPTY>
<!ATTLIST partref          refid IDREF #REQUIRED>

<!ELEMENT part             (name)>
<!ATTLIST part             part-id ID #REQUIRED
                           price CDATA #REQUIRED
                           source CDATA #REQUIRED>
]>

<parts-list>
  <component component-id="C28392-33-TT">
    <name>Turnip Twaddler</name>
    <partref refid="P81952-26-PK"/>
    <partref refid="P86679-52-SP"/>
    <partref refid="P81472-68-FD"/>
    <partref refid="P88107-39-GT"/>
  </component>
...
  <component component-id="C28772-63-OB">
    <name>Olive Bruiser</name>
    <partref refid="P80228-21-PT"/>
    <partref refid="P82387-85-PA"/>
  </component>

  <part part-id="P80228-21-PT" price="3.28" source="us">
    <name>Pitter</name>
  </part>
...
  <part part-id="P86994-25-RC" price="4.28" source="gb">
    <name>Ribbon Curler</name>
  </part>
</parts-list>
```

We'll use this stylesheet to resolve the references:

```
<?xml version="1.0"?>
<!-- id.xsl -->
<xsl:stylesheet version="2.0"
```

```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="text"/>

    <xsl:template match="/">
      <xsl:text>&#xA;Here is a test of the id() </xsl:text>
      <xsl:text>function:&#xA;</xsl:text>

      <xsl:for-each select="/parts-list/component">
        <xsl:sort select="name"/>
        <xsl:text>&#xA;  </xsl:text>
        <xsl:value-of select="name"/>
        <xsl:text> (part #</xsl:text>
        <xsl:value-of select="@component-id"/>
        <xsl:text>) uses these parts:&#xA;    </xsl:text>
        <xsl:for-each select="id(partref/@refid)">
          <xsl:value-of select="name"/>
          <xsl:text>&#xA;    </xsl:text>
        </xsl:for-each>
      </xsl:for-each>
    </xsl:template>
  </xsl:stylesheet>
```

Our stylesheet generates these results:

```
  Here is a test of the id() function:

    Clam Teaser (part #C28100-38-CT) uses these parts:
      Spanner
      Feather Duster
      Ribbon Curler

    Cucumber Decorating Kit (part #C23813-03-CDK) uses these parts:
      Feather Duster
      Paring Knife
      Mucilage
      Ribbon Curler

    Lemon Snubber (part #C21630-29-LS) uses these parts:
      Spanner
      Grommet
      Paring Knife

    Olive Bruiser (part #C28772-63-OB) uses these parts:
      Pitter
      Patter

    Prawn Goader (part #C28813-70-PG) uses these parts:
      Paring Knife
      Mucilage
      Ribbon Curler

    Turnip Twaddler (part #C28392-33-TT) uses these parts:
      Spanner
      Feather Duster
```

```
    Grommet
    Paring Knife
```

If the argument to the `id( )` function is a string, each space-separated token in the string is assumed to be a separate ID. That means we can use `id( )` with the datatype `IDREFS` as well. Here's a modified version of our source document that uses an `IDREFS` attribute:

```
<?xml version="1.0"?>
<!-- parts-list2.xsl -->
<!DOCTYPE parts-list [
<!ELEMENT parts-list       (component+, part+)>

<!ELEMENT component        (name, partref)>
<!ATTLIST component        component-id ID #REQUIRED>

<!ELEMENT name             (#PCDATA)>

<!ELEMENT partref          EMPTY>
<!ATTLIST partref          refid IDREFS #REQUIRED>

<parts-list>
  <component component-id="C28392-33-TT">
    <name>Turnip Twaddler</name>
    <partref
      refid="P81952-26-PK P86679-52-SP P81472-68-FD P88107-39-GT"/>
  </component>
...
  <part part-id="P80228-21-PT" price="3.28" source="us">
    <name>Pitter</name>
  </part>
...
</parts-list>
```

The only change here is that the `refid` attribute of the `<partref>` element is of datatype `IDREFS`. Despite that change, we can transform *parts-list2.xsl* with the same stylesheet and get the same results.

The value of any attribute of type `ID`, `IDREF`, and `IDREFS` must be valid XML names. A `ID` value of `86679-52-SP` is *not* a valid XML name, so an XSLT processor that validates the XML source document might not generate any results for that particular value. To see how this can cause problems, change all of the `ID`s so that they start with a number, and then run them through the Saxon XSLT processor. The results look like this:

```
Here is a test of the id() function:

  Turnip Twaddler (part #28392-33-TT) uses these parts:

  Prawn Goader (part #28813-70-PG) uses these parts:

  Clam Teaser (part #28100-38-CT) uses these parts:

  Cucumber Decorating Kit (part #23813-03-CDK) uses these parts:
```

```
Lemon Snubber (part #21630-29-LS) uses these parts:

Olive Bruiser (part #28772-63-OB) uses these parts:
```

When transforming the document with Saxon, we don't get any results. Using the Saxon `-v` flag to validate the XML source gives us dozens of messages like this:

```
Error on line 59 column 56 of file:/C:/XSLT20/AppendixC/parts-list3.xml:
  SXXP0003: Error reported by XML parser: Attribute value "80228-21-PT"
of type ID must be an NCName when namespaces are enabled.
```

Changing the XML source document so that all of the `ID`s are valid XML names gives us the results we expect.

Keep this in mind if you're unable to get the `id( )` *function working; it's a subtle error that can be hard to debug.* Using the current tools as of early 2008, Xalan and the Altova XML engine ignore the bad `ID`s, while MSXSL and Saxon don't generate any useful results.

[2.0] idref()

Given a sequence of `ID` values, returns a sequence containing all the nodes with an `IDREF` value matching one of the given `ID`s.

Syntax

```
node()* idref(xs:string*)
node()* idref(xs:string*, node())
```

Inputs

A sequence of `xs:string`s, each of which represents an `ID` value. If the `node( )` argument is supplied, the XSLT processor looks for matching `IDREF` and `IDREFS` values in the document that contains that node. Without the `node( )` argument, the processor looks in the document that contains the context node.

Outputs

A sequence of nodes, each of which has an `IDREF` or `IDREFS` value that matches one of the given `ID`s. The nodes are returned in document order, and any duplicate nodes are removed.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.5, "Functions and Operators that Generate Sequences."

Example

For our example, we'll use a list of components and parts with an embedded DTD. We'll list each part, followed by the name of each component that uses that part. The `idref( )` function lets us retrieve the `<component>` element with the specific `ID`; from there we can get the name of the component.

This is the exact opposite of the example for the `id( )` function. In that example, we take each `<component>` and list all of the parts that it uses.

To illustrate the `idref( )` function, we'll use an XML document with an embedded DTD. (This allows us to demonstrate the function without requiring a schema-aware XSLT processor.) Here is a fragment of the document:

```
<?xml version="1.0"?>
<!-- parts-list1.xml -->
<!DOCTYPE parts-list [
<!ELEMENT parts-list       (component+, part+)>

<!ELEMENT component        (name, partref+)>
<!ATTLIST component        component-id ID #REQUIRED>

<!ELEMENT name             (#PCDATA)>

<!ELEMENT partref          EMPTY>
<!ATTLIST partref          refid IDREF #REQUIRED>

<!ELEMENT part             (name)>
<!ATTLIST part             part-id ID #REQUIRED
                           price CDATA #REQUIRED
                           source CDATA #REQUIRED>
]>

<parts-list>
  <component component-id="C28392-33-TT">
    <name>Turnip Twaddler</name>
    <partref refid="P81952-26-PK"/>
    <partref refid="P86679-52-SP"/>
    <partref refid="P81472-68-FD"/>
    <partref refid="P88107-39-GT"/>
  </component>
...
  <component component-id="C28772-63-OB">
    <name>Olive Bruiser</name>
    <partref refid="P80228-21-PT"/>
    <partref refid="P82387-85-PA"/>
  </component>

  <part part-id="P80228-21-PT" price="3.28" source="us">
    <name>Pitter</name>
  </part>
...
  <part part-id="P86994-25-RC" price="4.28" source="gb">
    <name>Ribbon Curler</name>
  </part>
</parts-list>
```

Our document has a list of components; each component has an `component-id` attribute, a `<name>`, and a `<partrefs>` element. Each `<partrefs>` element has a `refids` attribute that refers

to some number <part> elements. The rules of the ID, IDREF, and IDREFS datatypes (or xs:ID, xs:IDREF, and xs:IDREFS in an XML Schema) are:

- Any attribute of type ID (xs:ID) must have a unique value. That value must be unique across all ID attributes. In other words, in our sample document, the component-id attribute of a <component> element can't have the same value as the part-id attribute of a <part> element.
- Any attribute of type IDREF (xs:IDREF) must have a value that matches an ID attribute somewhere in the document.
- Any attribute of type IDREFS (xs:IDREFS) must have a space-separated set of IDREF values.

Our stylesheet is pretty simple; it lists all the <part> items in the document, then it uses the idrefs( ) function to find all the <component>s that use that part. Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- idref.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Here is a test of the idref() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:for-each select="/parts-list/part">
      <xsl:sort select="name"/>
      <xsl:text>&#xA;  </xsl:text>
      <xsl:value-of select="name"/>
      <xsl:text> (part #</xsl:text>
      <xsl:value-of select="@part-id"/>
      <xsl:text>) is used in these products:&#xA;    </xsl:text>
      <xsl:value-of select="idref(@part-id)/../../name"
        separator="&#xA;    "/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

When we use the idref( ) function here, we pass in the value of the part-id attribute for the current <part> element. The idref( ) function returns the sequence of all the attributes of type IDREF whose values contain the current part-id.

Keep in mind that idref( ) returns a sequence of the attributes themselves, not their parents. That's why our XPath expression to print the value of each product is idref(@part-id)/../../name. The idref( ) function returns the sequence of the refid attribute nodes that match. To get the name of each product, we first get the refid's

parent and the <partref> element, then we get its parent and the <component> element, and finally we get its <name> child. Here are the results:

```
Here is a test of the idref() function:

  Feather Duster (part #P81472-68-FD) is used in these products:
    Turnip Twaddler
    Clam Teaser
    Cucumber Decorating Kit

  Grommet (part #P88107-39-GT) is used in these products:
    Turnip Twaddler
    Lemon Snubber

  Mucilage (part #P80499-43-MC) is used in these products:
    Prawn Goader
    Cucumber Decorating Kit

  Paring Knife (part #P81952-26-PK) is used in these products:
    Turnip Twaddler
    Prawn Goader
    Cucumber Decorating Kit
    Lemon Snubber

  Patter (part #P82387-85-PA) is used in these products:
    Olive Bruiser

  Pitter (part #P80228-21-PT) is used in these products:
    Olive Bruiser

  Ribbon Curler (part #P86994-25-RC) is used in these products:
    Prawn Goader
    Clam Teaser
    Cucumber Decorating Kit

  Spanner (part #P86679-52-SP) is used in these products:
    Turnip Twaddler
    Clam Teaser
    Lemon Snubber
```

One thing our sample document doesn't address is attributes of datatype IDREFS (xs:IDREFS). If we want, we could also design our <component> element like this:

```
<?xml version="1.0"?>
<!-- parts-list2.xml -->
<!DOCTYPE parts-list [
<!ELEMENT parts-list       (component+, part+)>

<!ELEMENT component        (name, partref)>
<!ATTLIST component        component-id ID #REQUIRED>

<!ELEMENT name             (#PCDATA)>

<!ELEMENT partref          EMPTY>
<!ATTLIST partref          refid IDREFS #REQUIRED>
```

```
<!ELEMENT part             (name) >
<!ATTLIST part             part-id ID #REQUIRED
                           price CDATA #REQUIRED
                           source CDATA #REQUIRED>
]>

<parts-list>
  <component component-id="C28392-33-TT">
    <name>Turnip Twaddler</name>
    <partref
      refid="P81952-26-PK P86679-52-SP P81472-68-FD P88107-39-GT"/>
  </component>
...
  <part part-id="P80228-21-PT" price="3.28" source="us">
    <name>Pitter</name>
  </part>
  <part part-id="P82387-85-PA" price="6.92" source="us">
    <name>Patter</name>
  </part>
...
</parts-list>
```

In this case, the `refid` attribute has a datatype of `IDREFS` (`xs:IDREFS`). That means the attribute has a whitespace-separated list of `IDREF` values. Because of the way `idref( )` works, we can use the stylesheet against this differently structured document and get the same results. The file *parts-list2.xml* changed how the references to the part IDs were structured, but it didn't change how or where the part IDs themselves were defined. That means the `idref( )` function still returns the results we want.

Using `idref( )`, we've retrieved all the uses of each of the `<part>`s in our inventory.

Keep in mind that values for `ID`, `IDREF`, and `IDREFS` datatypes must be valid XML names. `ID`s that start with a number can cause unpredictable results. See the note at the end of the discussion of the `id( )` function earlier in this appendix for all the details on this issue.

[2.0] implicit-timezone()

Returns the implicit timezone used when creating new `xs:date`, `xs:dateTime`, and `xs:time` values.

Syntax

```
xs:dayTimeDuration implicit-timezone()
```

Inputs

None.

Outputs

An `xs:dayTimeDuration` that contains the implicit timezone.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 16, "Context Functions."

Example

Here's a stylesheet that retrieves the implicit timezone as an `xs:dayTimeDuration`:

```
<?xml version="1.0"?>
<!-- implicit-timezone.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Using the implicit-timezone() function:&#xA;&#xA;</xsl:text>
    <xsl:text>  The implicit timezone for the current context is: </xsl:text>
    <xsl:value-of select="implicit-timezone()"/>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet generates these results:

```
Using the implicit-timezone() function:

  The implicit timezone for the current context is: -PT5H
```

The results for Bangalore and London, respectively, look like this:

```
  The implicit timezone for the current context is: PT5H30M
...
  The implicit timezone for the current context is: PT0S
```

[2.0] in-scope-prefixes()

Given an element, returns a sequence of all the namespace prefixes in scope for that element.

Syntax

```
xs:string* in-scope-prefixes(element())
```

Inputs

An element.

Outputs

A sequence of `xs:string`s listing the namespace prefixes that are in scope for that element.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 11.2, "Operators and Functions Related to QNames."

Example

To illustrate the `in-scope-prefixes( )` function, we'll reuse our Shakespearean sonnet:

```
<?xml version="1.0"?>
<!-- sonnet.xml -->
<sonnet type='Shakespearean'>
  <auth:author xmlns:auth="http://www.authors.com/">
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
    <year-of-death>1616</year-of-death>
  </auth:author>
  <!-- Is there an official title for this sonnet?  They're
       sometimes named after the first line.                    -->
  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
    <line>If hairs be wires, black wires grow on her head.</line>
    <line>I have seen roses damasked, red and white,</line>
    <line>But no such roses see I in her cheeks.</line>
    <line>And in some perfumes is there more delight</line>
    <line>Than in the breath that from my mistress reeks.</line>
    <line>I love to hear her speak, yet well I know</line>
    <line>That music hath a far more pleasing sound.</line>
    <line>I grant I never saw a goddess go,</line>
    <line>My mistress when she walks, treads on the ground.</line>
    <line>And yet, by Heaven, I think my love as rare</line>
    <line>As any she belied with false compare.</line>
  </lines>
</sonnet>
```

We'll use this stylesheet to list the namespace prefixes in scope for each element in the document:

```
<?xml version="1.0"?>
<!-- in-scope-prefixes.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the in-scope-prefixes() function:</xsl:text>

    <xsl:for-each select="//*">
      <xsl:text>&#xA;    In-scope prefixes for &lt;</xsl:text>
      <xsl:value-of select="name()"/>
```

```
      <xsl:text>&gt;: </xsl:text>
      <xsl:value-of select="in-scope-prefixes(.)" separator=", "/>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
A test of the in-scope-prefixes() function:
    In-scope prefixes for <sonnet>: xml
    In-scope prefixes for <auth:author>: xml, auth
    In-scope prefixes for <last-name>: xml, auth
    In-scope prefixes for <first-name>: xml, auth
    In-scope prefixes for <nationality>: xml, auth
    In-scope prefixes for <year-of-birth>: xml, auth
    In-scope prefixes for <year-of-death>: xml, auth
    In-scope prefixes for <title>: xml
    In-scope prefixes for <lines>: xml
    In-scope prefixes for <line>: xml
    In-scope prefixes for <line>: xml
    In-scope prefixes for <line>: xml
    In-scope prefixes for <line>: xml
    In-scope prefixes for <line>: xml
    In-scope prefixes for <line>: xml
    In-scope prefixes for <line>: xml
    In-scope prefixes for <line>: xml
    In-scope prefixes for <line>: xml
    In-scope prefixes for <line>: xml
    In-scope prefixes for <line>: xml
    In-scope prefixes for <line>: xml
    In-scope prefixes for <line>: xml
    In-scope prefixes for <line>: xml
```

The `xml` namespace prefix is always in scope; as the XSLT processor goes through the source document, the `auth` prefix goes in and out of scope.

[2.0] index-of()

Given a sequence and a search argument, returns a sequence of integers indicating the position(s) of the search argument in the sequence.

Syntax

```
xs:integer* index-of($sequenceParam as xs:anyAtomicType*,
                     $searchParam as xs:anyAtomicType)
xs:integer* index-of($sequenceParam as xs:anyAtomicType*,
                     $searchParam as xs:anyAtomicType,
                     $collation as xs:string)
```

Inputs

A sequence of atomic values and a search argument (also an atomic value). The optional third argument specifies the URL of a collation to be used when comparing the search argument to the values in the sequence.

Outputs

A sequence of `xs:integer`s representing the position(s) of the search argument in the sequence.

If the search sequence is the empty sequence or if no values in the sequence match the search argument, `index-of( )` returns the empty sequence. Also, when comparing the search argument to values in the sequence, the processor uses the rules defined for the `eq` operator for each datatype. If the `eq` operator is not defined for a given datatype, all values of that datatype will not be considered equal to the search argument.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.1, "General Functions and Operators on Sequences."

Example

Here is a short stylesheet that uses the `index-of( )` function:

```
<?xml version="1.0"?>
<!-- index-of.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:variable name="now" as="xs:time"
      select="current-time()"/>

    <xsl:variable name="testSequence1" as="item()*">
      <xsl:sequence
        select="(3, 4, 5, 'blue', $now, current-date(), 8, 'blue',
                 'Strasse', 'Stra&#xDF;e')"/>
    </xsl:variable>

    <xsl:text>&#xA;Here's a test of the index-of() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>&#xA;  Our first test sequence ($testSequence1) </xsl:text>
    <xsl:text>is:&#xA;      </xsl:text>
    <xsl:value-of select="$testSequence1" separator="&#xA;      "/>
    <xsl:text>&#xA;&#xA;</xsl:text>

    <xsl:text>    $now = </xsl:text>
    <xsl:value-of select="$now"/>
    <xsl:text>&#xA;&#xA;</xsl:text>
```

```
    <xsl:text>    index-of($testSequence1, 3) = (</xsl:text>
    <xsl:value-of
      select="index-of($testSequence1, 3)" separator=", "/>
    <xsl:text>)&#xA;&#xA;</xsl:text>

    <xsl:text>    index-of($testSequence1, 'red') = (</xsl:text>
    <xsl:value-of
      select="index-of($testSequence1, 'red')" separator=", "/>
    <xsl:text>)&#xA;&#xA;</xsl:text>

    <xsl:text>    index-of($testSequence1, 'blue') = (</xsl:text>
    <xsl:value-of
      select="index-of($testSequence1, 'blue')" separator=", "/>
    <xsl:text>)&#xA;&#xA;</xsl:text>

    <xsl:text>    index-of($testSequence1, $now) = (</xsl:text>
    <xsl:value-of
      select="index-of($testSequence1, $now)" separator=", "/>
    <xsl:text>)</xsl:text>

    <xsl:text>&#xA;&#xA;    index-of($testSequence1, </xsl:text>
    <xsl:text>'stra&#xDF;e') = (</xsl:text>
    <xsl:value-of
      select="index-of($testSequence1, 'Strasse')"
      separator=", "/>
    <xsl:text>)&#xA;</xsl:text>

    <xsl:text>&#xA;    index-of($testSequence1, </xsl:text>
    <xsl:text>'stra&#xDF;e', [German collation]) = (</xsl:text>
    <xsl:value-of
      select="index-of($testSequence1, 'Stra&#xDF;e',
              concat('http://saxon.sf.net/collation?',
                     'class=com.oreilly.xslt.GermanCollation;'))"
      separator=", "/>
    <xsl:text>)&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Here's a test of the index-of() function:

  Our first test sequence ($testSequence1) is:
      3
      4
      5
      blue
      15:04:18.841-05:00
      2006-11-23-05:00
      8
      blue
      Strasse
      Straße
```

```
$now = 15:04:18.841-05:00

index-of($testSequence1, 3) = (1)

index-of($testSequence1, 'red') = ()

index-of($testSequence1, 'blue') = (4, 8)

index-of($testSequence1, $now) = (5)

index-of($testSequence1, 'Strasse') = (9)

index-of($testSequence1, 'Strasse', [German collation]) = (9, 10)
```

Notice that we stored the current time in the variable `$now`. Technically we don't have to do this because the `current-time( )` function is stable; in other words, every time you call it during a single transformation, it returns the same result.

Also notice that we used a custom collation in the last call to `index-of( )`. In German, *Strasse* and *Straße* are two ways to spell the same word. The default collation doesn't recognize this, but our custom collation does. (To keep the listing within the margins of the page, we used the `concat( )` function to combine the two halves of the Saxon collation URI.) See "The document() Function and Sorting" in "The document() Function and Sorting for more information.

[2.0] insert-before()

Allows you to create a new sequence by inserting items into an existing sequence.

Syntax

```
item()* insert-before($target as item()*, $position as xs:integer,
                      $inserts as item()*)
```

Inputs

A sequence, an `xs:integer` (a position), and a second sequence.

Outputs

A new sequence in which the second sequence has been inserted into the first sequence before the item at the requested position.

The rules for `insert-before( )` are what you'd expect. If the first sequence is empty, the second sequence is returned unchanged. If the second sequence is empty, the first sequence is returned unchanged. If the position is less than `1`, the processor inserts the second sequence at the start of the first. Finally, if the position is greater than the number of items in the first sequence, the second sequence is inserted at the end of the first.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.1, "General Functions and Operators on Sequences."

Example

Here's a stylesheet that uses `insert-before( )` in several ways:

```
<?xml version="1.0"?>
<!-- insert-before.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:variable name="longSequence" as="item()*">
      <xsl:sequence
        select="(3, 4, 5, current-date(), current-time(), 8, 'blue')"/>
    </xsl:variable>

    <xsl:variable name="shortSequence" as="item()*"
      select="(current-dateTime(), xs:yearMonthDuration('P3Y8M'))"/>

    <xsl:text>&#xA;Here's a test of the insert-before() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>&#xA;  Our first sequence ($longSequence) </xsl:text>
    <xsl:text>is:&#xA;      </xsl:text>
    <xsl:value-of select="$longSequence" separator="&#xA;      "/>

    <xsl:text>&#xA;&#xA;  Our second sequence ($shortSequence) </xsl:text>
    <xsl:text>is:&#xA;      </xsl:text>
    <xsl:value-of select="$shortSequence" separator="&#xA;      "/>

    <xsl:text>&#xA;&#xA;    Test 1. insert-before($longSequence, </xsl:text>
    <xsl:text>3, (23)) </xsl:text>
    <xsl:text>= &#xA;      </xsl:text>
    <xsl:value-of select="insert-before($longSequence, 3, (23))"
      separator="&#xA;      "/>

    <xsl:text>&#xA;&#xA;    Test 2. insert-before($longSequence, </xsl:text>
    <xsl:text>4, $shortSequence) </xsl:text>
    <xsl:text>= &#xA;      </xsl:text>
    <xsl:value-of
      select="insert-before($longSequence, 4, $shortSequence)"
      separator="&#xA;      "/>

    <xsl:text>&#xA;&#xA;    Test 3. insert-before($longSequence, </xsl:text>
    <xsl:text>4, $shortSequence) </xsl:text>
    <xsl:text>= &#xA;      </xsl:text>
    <xsl:value-of
      select="insert-before($longSequence, 4, $shortSequence)"
      separator="&#xA;      "/>

    <xsl:text>&#xA;&#xA;    Test 4. insert-before($longSequence, 0, </xsl:text>
    <xsl:text>$shortSequence) </xsl:text>
    <xsl:text>= &#xA;      </xsl:text>
```

```
    <xsl:value-of select="insert-before($longSequence, 0, $shortSequence)"
      separator="&#xA;      "/>

    <xsl:text>&#xA;&#xA;    Test 5. insert-before($longSequence, 42, </xsl:text>
    <xsl:text>$shortSequence) </xsl:text>
    <xsl:text>= &#xA;      </xsl:text>
    <xsl:value-of select="insert-before($longSequence, 42, $shortSequence)"
      separator="&#xA;      "/>

    <xsl:text>&#xA;&#xA;    Creating a new variable based on </xsl:text>
    <xsl:text>&#xA;        </xsl:text>
    <xsl:text>insert-before($longSequence, 4, $shortSequence): </xsl:text>
    <xsl:text>&#xA;      </xsl:text>
    <xsl:text>updatedSequence = insert-before($longSequence, 4, </xsl:text>
    <xsl:text>$shortSequence) : &#xA;      </xsl:text>
    <xsl:variable name="updatedSequence" as="item()*"
      select="insert-before($longSequence, 4, $shortSequence)"/>
    <xsl:text>&#xA;    Test 6. $updatedSequence = &#xA;      </xsl:text>
    <xsl:value-of select="$updatedSequence" separator="&#xA;      "/>

  </xsl:template>

</xsl:stylesheet>
```

The results are:

```
Here's a test of the insert-before() function:

  Our first sequence ($longSequence) is:
      3
      4
      5
      2006-07-25-04:00
      04:24:21.216-04:00
      8
      blue

  Our second sequence ($shortSequence) is:
      2006-07-25T04:24:21.216-04:00
      P3Y8M

    Test 1. insert-before($longSequence, 3, (23)) =
      3
      4
      23
      5
      2006-07-25-04:00
      04:31:23.383-04:00
      8
      blue

    Test 2. insert-before($longSequence, 4, $shortSequence) =
      3
      4
      5
      2006-07-25T04:24:21.216-04:00
```

```
  P3Y8M
  2006-07-25-04:00
  04:24:21.216-04:00
  8
  blue

Test 3. insert-before($longSequence, 4, $shortSequence) =
  3
  4
  5
  2006-07-25T04:24:21.216-04:00
  P3Y8M
  2006-07-25-04:00
  04:24:21.216-04:00
  8
  blue

Test 4. insert-before($longSequence, 0, $shortSequence) =
  2006-07-25T04:24:21.216-04:00
  P3Y8M
  3
  4
  5
  2006-07-25-04:00
  04:24:21.216-04:00
  8
  blue

Test 5. insert-before($longSequence, 42, $shortSequence) =
  3
  4
  5
  2006-07-25-04:00
  04:24:21.216-04:00
  8
  blue
  2006-07-25T04:24:21.216-04:00
  P3Y8M

Creating a new variable based on
    insert-before($longSequence, 4, $shortSequence):
  updatedSequence = insert-before($longSequence, 4, $shortSequence) :

Test 6. $updatedSequence =
  3
  4
  5
  2006-07-25T04:24:21.216-04:00
  P3Y8M
  2006-07-25-04:00
  04:24:21.216-04:00
  8
  blue
```

We start the stylesheet by listing the two sequences we'll be using. For our first test, we hardcode a sequence (a singleton containing the value `23`) and insert it. Test 2 inserts the short sequence into the longer one before position `4` and returns the new sequence. Test 3 illustrates that calling `insert-before( )` doesn't change the original sequence; it merely returns a new sequence with the requested items inserted.

Tests 4 and 5 show what happens when the position is out of range. Specifying a position of `0` inserts the second sequence at the start of the first, while a position of `42` inserts the second sequence at the end of the first. Finally, test 6 stores the new sequence in a variable.

[2.0] iri-to-uri()

Given a string containing an Internationalized Resource Identifier (IRI), this function converts it to a URI. URI syntax allows only a subset of roughly 60 ASCII characters, which is inadequate for most of the world's languages. IRI syntax addresses this problem; this function translates IRI syntax to URI syntax.

Syntax

```
xs:string iri-to-uri(xs:string?)
```

Inputs

An `xs:string` containing an IRI.

Outputs

An `xs:string` with the appropriate characters escaped so that they are legal for a URI. (If you want all the details, see RFC 3987, Internationalized Resource Identifiers [IRIs], available at *http://www.ietf.org/rfc/rfc3987.txt*. For the details of URI syntax, see RFC 3986, Uniform Resource Identifiers [URI]: Generic Syntax, available at *http://www.ietf.org/rfc/rfc3986.txt*.)

This function does *not* escape the percent character (`%`). If you want to use this function, you must escape all the percent signs yourself (calling `replace-string($string, '%', '%25')` will do the trick) before you call `iri-to-uri( )`.

If the value of the argument is the empty sequence, a zero-length string is returned.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 7.4, "Functions on String Values."

Example

Here are two examples from the spec:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- iri-to-uri.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>
```

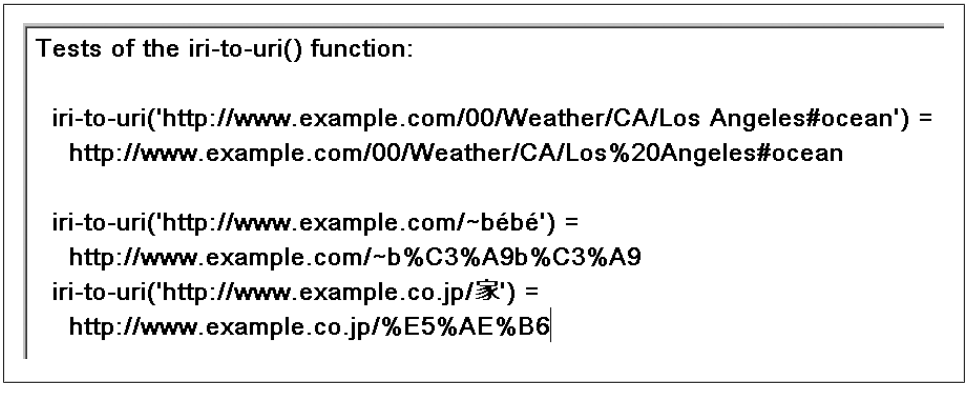


Figure C-4. Sample with illegal URI characters escaped

```
    <xsl:template match="/">
      <xsl:text>&#xA;Tests of the iri-to-uri() function:</xsl:text>

      <xsl:text>&#xA;&#xA;  iri-to-uri('http://www.example.com/</xsl:text>
      <xsl:text>00/Weather/CA/Los Angeles#ocean') = &#xA;    </xsl:text>
      <xsl:value-of
        select="iri-to-uri('http://www.example.com/00/Weather/CA/Los Angeles#ocean')"/>

      <xsl:text>&#xA;&#xA;  iri-to-uri('http://www.example.com/</xsl:text>
      <xsl:text>~b&#xE9;b&#xE9;') = &#xA;    </xsl:text>
      <xsl:value-of
        select="iri-to-uri('http://www.example.com/~b&#xE9;b&#xE9;')"/>

      <!-- The character &#x5BB6; is Japanese for "home," I hope... -->
      <xsl:text>&#xA;&#xA;  iri-to-uri('http://www.example.co.jp/</xsl:text>
      <xsl:text>&#x5BB6;') = &#xA;    </xsl:text>
      <xsl:value-of
        select="iri-to-uri('http://www.example.co.jp/&#x5BB6;')"/>
    </xsl:template>

</xsl:stylesheet>
```

As seen in Figure C-4, the IRIs are translated into URIs.

In the first example, the space was encoded as `%20`; in the second, the character *é* was encoded as the two-byte UTF-8 sequence `%C3%A9`.

The Japanese character in the last example (家) was correctly encoded as the three-byte UTF-8 sequence `%E5%AE%B6`. (I believe this is the character for "home," but I'm not 100% sure.)

See Also

The descriptions of the [2.0] `encode-for-uri( )` and [2.0] `escape-html-uri( )` functions.

key()

References a relation defined with an <xsl:key> element. Conceptually, the key( ) function works similarly to the id( ) function, although keys are more flexible than IDs.

Syntax

```
[1.0] node-set key(string, object)
[2.0] node()* key(xs:string, xs:anyAtomicType*)
[2.0] node()* key(xs:string, xs:anyAtomicType*, node())
```

Inputs

[1.0] The name of the key (defined by an <xsl:key> element) and an object. If the object is a node-set, then the key( ) function applies itself to the string value of each node in the node-set and returns the node-set of the result of all those key( ) function invocations. If the object is any other type, it is converted to a string as if by a call to the string( ) function.

[2.0] The name of the key (an xs:string) and a sequence of search values. An optional third argument limits the search to all the nodes that have the specified node as an ancestor-or-self node. The third argument lets us limit the search results to a particular group of nodes.

Output

[1.0] A node-set containing the nodes in the same document as the context node whose values for the requested key match the search argument(s). In other words, if our stylesheet has an <xsl:key> element that defines a key named postalcodes based on the <postalcode> child of all <address> elements in the current document, the function call key(postalcodes, '34829') returns a node-set containing all the <address> elements with a <postalcode> element whose value is 34829.

[2.0] A sequence of nodes, each of which matches one of the values in the sequence of search values. How those nodes match the search arguments is determined by the specified key.

[2.0] Keep in mind that any untyped values in the sequence of search values will be compared as strings. If the key you're using defined xs:date values as the index property, your string values will never match and key( ) will always return the empty sequence. In this case, only search values of type xs:date will return any results.

Defined in

[1.0] XSLT section 12.2, "Keys."

[2.0] XSLT section 16.3, "Keys."

Example

To illustrate the power of the key( ) function, we'll use this document:

```
<?xml version="1.0"?>
<!-- simplified-names.xml -->
```

```
<addressbook>
  <address>
    <first-name>Chester Hasbrouck</first-name>
    <last-name>Frisby</last-name>
    <city>Sheboygan</city>
    <state>WI</state>
  </address>
  <address>
    <first-name>Mary</first-name>
    <last-name>Backstayge</last-name>
    <city>Skunk Haven</city>
    <state>MA</state>
  </address>
  <address>
    <first-name>Natalie</first-name>
    <last-name>Attired</last-name>
    <city>Winter Harbor</city>
    <state>ME</state>
  </address>
  <address>
    <first-name>Harry</first-name>
    <last-name>Backstayge</last-name>
    <city>Skunk Haven</city>
    <state>MA</state>
  </address>
  <address>
    <first-name>Mary</first-name>
    <last-name>McGoon</last-name>
    <city>Boylston</city>
    <state>VA</state>
  </address>
  <address>
    <first-name>Amanda</first-name>
    <last-name>Reckonwith</last-name>
    <city>Lynn</city>
    <state>MA</state>
  </address>
</addressbook>
```

Here's the stylesheet we'll use to process this document. Notice that we define a key to index customer addresses by state:

```
<?xml version="1.0"?>
<!-- key-function.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:param name="searchState"/>

  <xsl:key name="customersByState" match="address" use="state"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the key() function:&#xA;</xsl:text>
    <xsl:text>&#xA;  All customers in </xsl:text>
```

```
    <xsl:value-of select="$searchState"/>
    <xsl:text>&#xA;&#xA;    </xsl:text>
    <xsl:for-each select="key('customersByState', $searchState)">
      <xsl:value-of select="first-name"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="last-name"/>
      <xsl:text>, </xsl:text>
      <xsl:value-of select="city"/>
      <xsl:text>&#xA;    </xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Running our stylesheet with a `$searchState` parameter of `MA` gives these results:

```
A test of the key() function:

  All customers in MA

    Mary Backstayge, Skunk Haven
    Harry Backstayge, Skunk Haven
    Amanda Reckonwith, Lynn
```

A search parameter of `WI` finds only one customer:

```
A test of the key() function:

  All customers in WI

    Chester Hasbrouck Frisby, Sheboygan
```

lang()

Determines whether a given language string is the same as, or is a sublanguage of, the language of the context node, as defined by an `xml:lang` attribute.

Syntax

```
[1.0] boolean lang(string)
[2.0] xs:boolean lang(xs:string?)
[2.0] xs:boolean lang(xs:string?, node())
```

Inputs

A string representing a language code. If the context node has a language of `xml:lang="en-us"`, invoking the `lang( )` function with any of the values `en`, `EN`, and `en-us` returns the boolean value `true`, while invoking `lang( )` with the value `en-gb` returns the boolean value `false`.

Output

If the argument string is the same as, or is a sublanguage of, the context node's language, `lang( )` returns the boolean value `true`. If the context node does not have an `xml:lang` attribute, then the value of the `xml:lang` attribute of its nearest ancestor is used instead. If there is no such attribute, then the `lang( )` function returns the boolean value `false`. When comparing

the language code of the context node with the argument string, the `lang( )` function ignores case.

Defined in

[1.0] XPath section 4.3, "Boolean Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 14, "Functions and Operators on Nodes."

Example

Here is an XML document that uses language codes:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

Here's a stylesheet that uses the `lang( )` function:

```
<?xml version="1.0"?>
<!-- lang.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the lang() function:&#xA;&#xA;</xsl:text>
    <xsl:for-each select="list/listitem">
      <xsl:choose>
        <xsl:when test="lang('EN')">
          <xsl:text>  Here's an English-language album: </xsl:text>
        </xsl:when>
        <xsl:otherwise>
          <xsl:text>  -------> Here's some World music: </xsl:text>
        </xsl:otherwise>
      </xsl:choose>
      <xsl:value-of select="."/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Finally, here are the results:

```
A test of the lang() function:

  Here's an English-language album: The Sacred Art of Dub
  Here's an English-language album: Only the Poor Man Feel It
  Here's an English-language album: Excitable Boy
  -------> Here's some World music: Aki Special
  Here's an English-language album: Combat Rock
  -------> Here's some World music: Talking Timbuktu
  -------> Here's some World music: The Birth of the Cool
```

Notice that the search argument of `EN` matches both `en`, the default language, and `en-gb`, the code for U.K. English.

last()

Returns the position of the last node in the current context. This function is useful for defining templates for the last occurrence of a given element or for testing whether a given node is the last in the node-set to which it belongs.

Syntax

[1.0] number **last()**
[2.0] xs:integer **last()**

Inputs

None.

Output

A number equal to the number of nodes in the current context. For example, if the current context contains 12 `<li>` nodes, `last( )` returns 12.

Defined in

[1.0] XPath section 4.1, "Node Set Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 16, "Context Functions."

Example

We'll use the `last( )` function to handle the last item in a list in a special way. Here's the XML document we'll use:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
```

```
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

Here is the stylesheet that handles the last `<listitem>` in the list differently:

```
<?xml version="1.0"?>
<!-- last.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the last() function:&#xA;</xsl:text>
    <xsl:for-each select="list/listitem">
      <xsl:text>&#xA;  </xsl:text>
      <xsl:if test="position()=last()">
        <xsl:text>&#xA;  LAST, but not least:  </xsl:text>
      </xsl:if>
      <xsl:value-of select="."/>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

When we transform the XML document with this stylesheet, here are the results:

```
A test of the last() function:

  The Sacred Art of Dub
  Only the Poor Man Feel It
  Excitable Boy
  Aki Special
  Combat Rock
  Talking Timbuktu

  LAST, but not least:  The Birth of the Cool
```

local-name()

Returns the local part of the first node in the argument node-set.

Syntax

```
[1.0] string local-name(node-set?)
[2.0] xs:string local-name()
[2.0] xs:string local-name(node()?)
```

Inputs

A node-set. If the node-set is empty, the function returns an empty string. If the node-set is omitted, the function uses a node-set with the context node as its only member.

Output

A string corresponding to the local name of the first element in the argument node-set. If the node-set is empty, the `local-name( )` function returns an empty string.

[2.0] In XSLT 2.0, it is a fatal error if the argument contains more than one node.

Defined in

[1.0] XPath section 4.1, "Node Set Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 14, "Functions and Operators on Nodes."

Example

We'll use our Shakespearean sonnet to test the `local-name( )` function:

```
<?xml version="1.0"?>
<!-- sonnet.xml -->
<sonnet type='Shakespearean'>
  <auth:author xmlns:auth="http://www.authors.com/">
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
    <year-of-death>1616</year-of-death>
  </auth:author>
  <!-- Is there an official title for this sonnet?  They're
       sometimes named after the first line.                  -->
  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
    <line>If hairs be wires, black wires grow on her head.</line>
    <line>I have seen roses damasked, red and white,</line>
    <line>But no such roses see I in her cheeks.</line>
    <line>And in some perfumes is there more delight</line>
    <line>Than in the breath that from my mistress reeks.</line>
    <line>I love to hear her speak, yet well I know</line>
    <line>That music hath a far more pleasing sound.</line>
    <line>I grant I never saw a goddess go,</line>
    <line>My mistress when she walks, treads on the ground.</line>
    <line>And yet, by Heaven, I think my love as rare</line>
    <line>As any she belied with false compare.</line>
  </lines>
</sonnet>
```

Here's the short stylesheet we'll use:

```
<?xml version="1.0"?>
<!-- local-name.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>
```

```
  <xsl:template match="/">
    <xsl:text>&#xA;A test of the local-name() function:&#xA;</xsl:text>

    <xsl:for-each select="//*">
      <xsl:text>&#xA;    The local name for &lt;</xsl:text>
      <xsl:value-of select="name()"/>
      <xsl:text>&gt;: </xsl:text>
      <xsl:value-of select="local-name(.)"/>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results from the stylesheet:

```
A test of the local-name() function:

    The local name for <sonnet>: sonnet
    The local name for <auth:author>: author
    The local name for <last-name>: last-name
    The local name for <first-name>: first-name
    The local name for <nationality>: nationality
    The local name for <year-of-birth>: year-of-birth
    The local name for <year-of-death>: year-of-death
    The local name for <title>: title
    The local name for <lines>: lines
    The local name for <line>: line
    The local name for <line>: line
    The local name for <line>: line
    The local name for <line>: line
    The local name for <line>: line
    The local name for <line>: line
    The local name for <line>: line
    The local name for <line>: line
    The local name for <line>: line
    The local name for <line>: line
    The local name for <line>: line
    The local name for <line>: line
    The local name for <line>: line
    The local name for <line>: line
    The local name for <line>: line
```

[2.0] local-name-from-QName()

Given an `xs:QName` value, returns its local name.

Syntax

```
xs:NCName? local-name-from-QName(xs:QName?)
```

Input

An `xs:QName` value.

Output

The local name portion of the given `QName`. This is returned as an `xs:NCName`, or "noncolonized" name. If the argument is the empty sequence, `local-name-from-QName( )` returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 11.2, "Operators and Functions Related to QNames."

Example

Here is a short stylesheet that uses `local-name-from-QName( )`:

```
<?xml version="1.0"?>
<!-- local-name-from-qname.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the local-name-from-QName() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:variable name="testQName1" as="xs:QName"
      select="QName('http://www.authors.com', 'auth:author')"/>

    <xsl:text>&#xA;  QName('http://www.authors.com', </xsl:text>
    <xsl:text>'auth:author') = </xsl:text>
    <xsl:value-of select="$testQName1"/>

    <xsl:text>&#xA;    The local name associated with </xsl:text>
    <xsl:text>this QName: "</xsl:text>
    <xsl:value-of select="local-name-from-QName($testQName1)"/>
    <xsl:text>"</xsl:text>

    <xsl:variable name="testQName2" as="xs:QName"
      select="QName('', 'sonnet')"/>

    <xsl:text>&#xA;&#xA;  QName('', 'sonnet') = </xsl:text>
    <xsl:value-of select="$testQName2"/>

    <xsl:text>&#xA;    The local name associated with </xsl:text>
    <xsl:text>this QName: "</xsl:text>
    <xsl:value-of select="local-name-from-QName($testQName2)"/>
    <xsl:text>"</xsl:text>

  </xsl:template>

</xsl:stylesheet>
```

The straightforward results look like this:

```
Tests of the local-name-from-QName() function:

  QName('http://www.authors.com', 'auth:author') = auth:author
    The local name associated with this QName: "author"

  QName('', 'sonnet') = sonnet
    The local name associated with this QName: "sonnet"
```

[2.0] lower-case()

Given a string, returns the lowercased version of that string.

Syntax

```
xs:string lower-case(xs:string)
```

Inputs

An `xs:string` value.

Outputs

An `xs:string` in which all of the uppercase letters in the original string have been converted to lowercase. Any character that was originally in lowercase and any character that does not have an lowercase value is returned as is. If the value of the argument is the empty sequence, a zero-length string is returned.

Accented characters and other features of the world's languages mean that changing the case of a string might change its length. Also be aware that `lower-case( )` and `upper-case( )` are not always the inverse of each other in some languages. All of the case conversion rules are defined by the Unicode standard, and XSLT processors are expected to conform with those rules.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 7.4, "Functions on String Values."

Example

Here is a stylesheet that illustrates the `lower-case( )` function. Notice that we're using `<xsl:output method="xml" encoding="UTF-8"/>` to make sure the character set is handled properly:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- lower-case.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" encoding="UTF-8" indent="yes"/>

  <xsl:template match="/">
    <testcase>
```

```
      <heading>Tests of the lower-case() function:</heading>
      <test>
        <label>lower-case('Lily') = </label>
        <result><xsl:value-of select="lower-case('Lily')"/></result>
      </test>
      <test>
        <label>lower-case('LILY') = </label>
        <result><xsl:value-of select="lower-case('LILY')"/></result>
      </test>
      <test>
        <label>lower-case('lily') = </label>
        <result><xsl:value-of select="lower-case('lily')"/></result>
      </test>
      <test>
        <label>lower-case('JALAPEÑO') = </label>
        <result><xsl:value-of select="lower-case('JALAPEÑO')"/></result>
      </test>
    </testcase>
  </xsl:template>

</xsl:stylesheet>
```

The results look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<testcase>
   <heading>Tests of the lower-case() function:</heading>
   <test>
      <label>lower-case('Lily') = </label>
      <result>lily</result>
   </test>
   <test>
      <label>lower-case('LILY') = </label>
      <result>lily</result>
   </test>
   <test>
      <label>lower-case('lily') = </label>
      <result>lily</result>
   </test>
   <test>
      <label>lowercase('JALAPEÑO') = </label>
      <result>jalapeño</result>
   </test>
</testcase>
```

[2.0] matches()

Determines whether a given string matches a given regular expression.

Syntax

```
xs:boolean matches($input as xs:string?, $pattern as xs:string)
xs:boolean matches($input as xs:string?, $pattern as xs:string,
                   $flags as xs:string)
```

Inputs

An input string and a regular expression. The `matches( )` function also supports an optional string defining flags that modify how the regular expression is processed.

Outputs

`true` if the string matches the regular expression; `false` otherwise.

Regular expressions are extremely powerful, so there are a number of details about how they work:

- If `$input` is the empty sequence, it is interpreted as a zero-length string. The `matches( )` function can still return `true` in this situation. For example, `matches(( ), '.?')` is `true`, as is `matches('', '.?')`.
- Regular expression matching does not use collations; the characters' Unicode code points are compared. Cases in which different characters are considered equal in the world's languages are not taken into account.
- Unless you use the ^ (start of line) or `$` (end of line) characters as anchors in your regular expression, a string is considered a match if any part of it matches the regular expression.
- Unlike regular expressions used in the `<xsl:analyze-string>` element, curly braces ({ and }) are not doubled. (Curly braces used inside the `regex` attribute of `<xsl:analyze-string>` must be doubled so they aren't interpreted as attribute value templates.)
- The `$flags` parameter modifies how the regular expression is processed. There are four different flags:

  `s`

  Regular expressions are evaluated in what the specs refer to as "dot-all" mode. When this flag is used, the dot operator (`.`) matches any character. Under normal processing (without the `s` flag), the dot operator matches any character *except* the newline character (`
`). This flag is useful when you want to match strings that might include a newline character.

  `m`

  Regular expressions are evaulated in multiline mode. By default, the meta-character (^) matches the start of the entire string, while $ matches the end of the entire string. In multiline mode, ^ matches the start of any line within the string and $ matches the end of any line within the string.

  `i`

  Regular expressions are evaluated in case-insensitive mode. The regular expression `"a"` matches both `"a"` and `"A"`.

  Note that Unicode issues can complicate this greatly. For example, the XQuery 1.0 and XPath 2.0 Functions and Operators spec gives the example of the Unicode sign for degrees Kelvin (`K`), which is the letter `"K"`. The

combination of `regex="k"` and `flags="i"` matches the Kelvin sign as well as the letters `"k"` (`k`) and `"K"` (`K`).

Other Unicode characters don't convert to letters. For example, the Unicode symbol for the Roman numeral `I` (`ࡰ`) looks like the letter `I`, but does not convert to one.

`x`

All whitespace characters (`	`, `
`, `
`, and ` `) are removed from the regular expression before any comparison is done. In other words, with the `x` flag, the regular expressions `"John Smith"` and `"JohnSmith"` are the same. This flag is useful when you want to break a long regular expression into multiple lines to make it easier to read.

The flags can be combined in any order. The parameters `'xis'` and `'six'` work exactly the same way.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 7.6, "String Functions that Use Pattern Matching."

Example

Here is a stylesheet that uses the `matches( )` function:

```
<?xml version="1.0"?>
<!-- matches.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:variable name="string1"
      select="concat('Now is the time for all good men&#xA;',
              'and women &#xA;',
              'to aid the party.')"/>

    <xsl:text>&#xA;Here's a test of the matches() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>&#xA;  $string1 = &#xA;</xsl:text>
    <xsl:value-of select="$string1"/>

    <xsl:text>&#xA;&#xA;  Test 1. matches($string1, </xsl:text>
    <xsl:text>'Now.*men') = </xsl:text>
    <xsl:value-of select="matches($string1, 'Now.*men')"/>

    <xsl:text>&#xA;&#xA;  Test 2. matches($string1, </xsl:text>
    <xsl:text>'Now.*men$') = </xsl:text>
    <xsl:value-of select="matches($string1, 'Now.*men$')"/>
```

```
    <xsl:text>&#xA;&#xA;  Test 3. matches($string1, </xsl:text>
    <xsl:text>'Now.*men$', 'm') = </xsl:text>
    <xsl:value-of select="matches($string1, 'Now.*men$', 'm')"/>

    <xsl:text>&#xA;&#xA;  Test 4. matches($string1, </xsl:text>
    <xsl:text>'women $', 'm') = </xsl:text>
    <xsl:value-of select="matches($string1, 'women $', 'm')"/>

    <xsl:text>&#xA;&#xA;  Test 5. matches($string1, </xsl:text>
    <xsl:text>'party.$') = </xsl:text>
    <xsl:value-of select="matches($string1, 'party.$')"/>

    <xsl:text>&#xA;&#xA;  Test 6. matches($string1, </xsl:text>
    <xsl:text>'^and' 'm') = </xsl:text>
    <xsl:value-of select="matches($string1, '^and', 'm')"/>

    <xsl:text>&#xA;&#xA;  Test 7. matches($string1, </xsl:text>
    <xsl:text>'women .?to' 's') = </xsl:text>
    <xsl:value-of select="matches($string1, 'women .?to', 's')"/>

  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Here's a test of the matches() function:

  $string1 =
Now is the time for all good men
and women
to aid the party.

  Test 1. matches($string1, 'Now.*men') = true

  Test 2. matches($string1, 'Now.*men$') = false

  Test 3. matches($string1, 'Now.*men$', 'm') = true

  Test 4. matches($string1, 'women $', 'm') = true

  Test 5. matches($string1, 'party.$') = true

  Test 6. matches($string1, '^and' 'm') = true

  Test 7. matches($string1, 'women .?to' 's') = true
```

Our sample string contains two line breaks so we can test the `^` and `$` characters along with the flags. The regular expression in test 1 is the characters `'Now'` followed by zero or more characters, followed by the characters `'men'`. Test 2 specifies that `'men'` should be the last three characters in the string; that's clearly not the case here. In test 3, we specify that the expression should be processed in multiline mode (using the `'m'` flag), so `'men
'` is considered a match for `'men$'`.

In test 4, we look for the characters `'women '`, at the end of a line. Because we're using multiline mode, this matches. Notice that we had to include the space character at the end of the line; `'women$'` does not match here. Test 5 specifies the characters `'party.'` at the end of the string. We're not in multiline mode, but that's OK; the last characters in the string are `'party.'` Test 6 looks for the characters `'and'` at the start of a line; because we're in multiline mode, this works.

Finally, we look for the characters `'women '` followed by a single optional character followed by the characters `'to'`. We're using "dot all" mode here (we specified the `'s'` flag), so this is a match. The single optional character here is a newline, which matches when the `'s'` flag is in effect.

See Also

Appendix E has complete details on the way regular expressions work in XPath 2.0. Also see the definitions of the following elements and functions: [2.0] `<xsl:analyze-string>`, [2.0] `<xsl:matching-substring>`, [2.0] `<xsl:non-matching-substring>`, [2.0] `regex-group( )`, [2.0] `replace( )`, and [2.0] `tokenize( )`.

[2.0] max()

Returns the largest value in a given sequence.

Syntax

```
xs:anyAtomicType max(xs:anyAtomicType*)
xs:anyAtomicType max(xs:anyAtomicType*, $collation as xs:string)
```

Input

A sequence of values.

Output

The maximum value in the given sequence. The `max( )` function works with the following types of values: numeric values (`xs:integer`, `xs:double`, `xs:decimal`, and `xs:float`), `xs:string`, `xs:boolean`, `xs:date`, `xs:dateTime`, `xs:time`, `xs:yearMonthDuration`, and `xs:dayTimeDuration` (but *not* `xs:duration`).

Given a sequence of numeric values, the XSLT processor returns the largest number, converting datatypes as necessary. Given a sequence of durations, the XSLT processor returns the longest of those durations. Given a sequence of strings, the XSLT processor returns the string that appears last in sorted order.

The `max( )` function assumes you'll send it a sequence containing sensible data; if not, the XSLT processor throws an error. Asking for the maximum value in the sequence `(xs:dayTimeDuration('P3DT8H17M'), 38, 'football')` returns an error, as you'd expect.

Details about how the `max( )` function works are as follows:

- To find the longest of a sequence of durations, the values must all be `xs:dayTimeDuration`s, or they must all be `xs:yearMonthDuration`s. You can't mix the two types of durations; if you do, the XSLT processor throws an error.
- If any of the items in the sequence are of type `xs:untypedAtomic`, the XSLT processor attempts to cast it to `xs:double`. If the item can't be converted to `xs:double`, the XSLT processor throws an error.
- When comparing `xs:string` values, the default collation is used unless you specify another.
- Finally, if you pass the `max( )` function the empty sequence, it returns the empty sequence. Although you're not giving the function any useful data in this case, the XSLT processor doesn't throw an error.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.4, "Aggregate Functions."

Example

Here's a stylesheet that finds the maximum value in several different types of sequences:

```
<?xml version="1.0"?>
<!-- max.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="seq1" select="(3, 5, 18)"/>
    <xsl:variable name="seq2" select="(3, 5, 48.273, 2.9e3)"/>

    <xsl:variable name="value1" as="xs:integer" select="42"/>
    <xsl:variable name="value2" as="xs:double" select="2718.28E-3"/>
    <xsl:variable name="value3" as="xs:float" select="98.6"/>
    <xsl:variable name="value4" as="xs:decimal" select="2.54"/>
    <xsl:variable name="seq3"
      select="($value1, $value2, $value3, $value4)"/>

    <xsl:variable name="seq4"
      select="(xs:yearMonthDuration('P3Y8M'),
               xs:yearMonthDuration('P4Y2M'),
               xs:yearMonthDuration('P6Y4M'))"/>
    <xsl:variable name="seq5"
      select="(xs:dayTimeDuration('P2DT4H23M12.2S'),
               xs:dayTimeDuration('P3DT8H17M'),
               xs:dayTimeDuration('P3D'))"/>

    <xsl:text>&#xA;Here are some tests of the max() function:&#xA;</xsl:text>

    <xsl:text>&#xA;  max(</xsl:text>
    <xsl:value-of select="$seq1" separator=", "/>
```

```
<xsl:text>) = </xsl:text>
<xsl:value-of select="format-number(max($seq1), '#.###')"/>

<xsl:text>&#xA;&#xA;  max(</xsl:text>
<xsl:value-of select="$seq2" separator=", "/>
<xsl:text>) = </xsl:text>
<xsl:value-of select="format-number(max($seq2), '#.###')"/>

<xsl:text>&#xA;&#xA;  max(</xsl:text>
<xsl:value-of select="$seq3" separator=", "/>
<xsl:text>) = </xsl:text>
<xsl:value-of select="format-number(max($seq3), '#.###')"/>

<xsl:text>&#xA;&#xA;  max(</xsl:text>
<xsl:value-of select="$seq4" separator=", "/>
<xsl:text>) = </xsl:text>
<xsl:value-of select="max($seq4)"/>

<xsl:text>&#xA;&#xA;    In text, the maximum of</xsl:text>
<xsl:for-each select="$seq4">
  <xsl:text>&#xA;      </xsl:text>
  <xsl:value-of select="years-from-duration(.)"/>
  <xsl:text> years and </xsl:text>
  <xsl:value-of select="months-from-duration(.)"/>
  <xsl:text> months (</xsl:text>
  <xsl:value-of select="."/>
  <xsl:text>)</xsl:text>
</xsl:for-each>

<xsl:text>&#xA;    is </xsl:text>
<xsl:value-of select="years-from-duration(max($seq4))"/>
<xsl:text> years and </xsl:text>
<xsl:value-of select="months-from-duration(max($seq4))"/>
<xsl:text> months (</xsl:text>
<xsl:value-of select="max($seq4)"/>
<xsl:text>).</xsl:text>

<xsl:text>&#xA;&#xA;  max(</xsl:text>
<xsl:value-of select="$seq5" separator=", "/>
<xsl:text>) = </xsl:text>
<xsl:variable name="max5" select="max($seq5)"/>
<xsl:value-of select="$max5"/>

<xsl:text>&#xA;&#xA;    In text, the maximum of</xsl:text>
<xsl:for-each select="$seq5">
  <xsl:text>&#xA;      </xsl:text>
  <xsl:value-of select="days-from-duration(.)"/>
  <xsl:text> days, </xsl:text>
  <xsl:value-of select="hours-from-duration(.)"/>
  <xsl:text> hours, </xsl:text>
  <xsl:value-of select="minutes-from-duration(.)"/>
  <xsl:text> minutes and </xsl:text>
  <xsl:value-of
    select="format-number(seconds-from-duration(.), '#.##')"/>
  <xsl:text> seconds (</xsl:text>
```

```
      <xsl:value-of select="."/>
      <xsl:text>)</xsl:text>
    </xsl:for-each>

    <xsl:text>&#xA;    is </xsl:text>
    <xsl:value-of select="days-from-duration($max5)"/>
    <xsl:text> days, </xsl:text>
    <xsl:value-of select="hours-from-duration($max5)"/>
    <xsl:text> hours, </xsl:text>
    <xsl:value-of select="minutes-from-duration($max5)"/>
    <xsl:text> minutes and </xsl:text>
    <xsl:value-of
      select="format-number(seconds-from-duration($max5), '#.##')"/>
    <xsl:text> seconds.</xsl:text>

  </xsl:template>

</xsl:stylesheet>
```

Here are the results from this stylesheet:

```
Here are some tests of the max() function:

  max(3, 5, 18) = 18

  max(3, 5, 48.273, 2900) = 2900

  max(42, 2.71828, 98.6, 2.54) = 98.6

  max(P3Y8M, P4Y2M, P6Y4M) = P6Y4M

    In text, the maximum of
      3 years and 8 months (P3Y8M)
      4 years and 2 months (P4Y2M)
      6 years and 4 months (P6Y4M)
    is 6 years and 4 months (P6Y4M).

  max(P2DT4H23M12.2S, P3DT8H17M, P3D) = P3DT8H17M

    In text, the maximum of
      2 days, 4 hours, 23 minutes and 12.2 seconds (P2DT4H23M12.2S)
      3 days, 8 hours, 17 minutes and 0 seconds (P3DT8H17M)
      3 days, 0 hours, 0 minutes and 0 seconds (P3D)
    is 3 days, 8 hours, 17 minutes and 0 seconds.
```

The stylesheet demonstrates five different types of sequences. The first is all integers and the second is a sequence of numbers. All the values in the first two sequences can be converted to numbers, so the results are what we expect. The third sequence features four numeric values, each of which has a specific numeric type. The last two sequences are made up of the two types of durations. The stylesheet also uses functions such as `years-from-duration( )` and `format-number( )` to format the data.

Here's a short example that illustrates using a custom collation:

```
<?xml version="1.0"?>
<!-- max2.xsl -->
```

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="seq1"
      select="('strasse', 'stra&#xDF;e')"/>
    <xsl:variable name="seq2" select="reverse($seq1)"/>

    <xsl:text>&#xA;Here are some tests of the max() function:&#xA;</xsl:text>

    <xsl:text>&#xA;  max(</xsl:text>
    <xsl:value-of select="$seq1" separator=", "/>
    <xsl:text>) = </xsl:text>
    <xsl:value-of select="max($seq1)"/>

    <xsl:text>&#xA;  max(</xsl:text>
    <xsl:value-of select="$seq1" separator=", "/>
    <xsl:text> [German collation]) = </xsl:text>
    <xsl:value-of
      select="max($seq1,
              concat('http://saxon.sf.net/collation?',
                     'class=com.oreilly.xslt.GermanCollation;'))"/>

    <xsl:text>&#xA;  max(</xsl:text>
    <xsl:value-of select="$seq2" separator=", "/>
    <xsl:text> [German collation]) = </xsl:text>
    <xsl:value-of
      select="max($seq2,
              concat('http://saxon.sf.net/collation?',
                     'class=com.oreilly.xslt.GermanCollation;'))"/>

  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Here are some tests of the max() function:

  max(strasse, straße) = straße
  max(strasse, straße [German collation]) = strasse
  max(straße, strasse [German collation]) = straße
```

Using the default collation, these two strings are not equal. In the first example, `straße` sorts after `strasse`, so that's the value returned by the `max( )` function. In the final two examples, the two strings are considered equal by the German collation. In Saxon 9.0.0.3J, `max( )` returns the maximum value that appears first in the sequence. This is implementation dependent; XSLT processors are allowed to return any of the equal maximum values.

[2.0] min()

Returns the smallest value in a given sequence.

Syntax

```
xs:anyAtomicType min(xs:anyAtomicType*)
xs:anyAtomicType min(xs:anyAtomicType*, $collation as xs:string)
```

Input

A sequence of values. The optional collation is only used if the items in the sequence are `xs:string`s.

Output

The minimum value in the given sequence. The `min( )` function works with the following types of values: numeric values (`xs:integer`, `xs:double`, `xs:decimal`, and `xs:float`), `xs:string`, `xs:boolean`, `xs:date`, `xs:dateTime`, `xs:time`, `xs:yearMonthDuration`, and `xs:dayTimeDuration` (but *not* `xs:duration`).

Given a sequence of numeric values, the XSLT processor returns the smallest number, converting datatypes as necessary. Given a sequence of durations, the XSLT processor returns the shortest of those durations. Given a sequence of strings, the XSLT processor returns the string that appears first in sorted order.

The `min( )` function assumes you'll send it a sequence containing sensible data; if not, the XSLT processor throws an error. Asking for the minimum value in the sequence `(xs:yearMonthDuration('P4Y2M'), 'strawberry', 3.14)` returns an error, as you'd expect.

Some notes about how the `min( )` function works are as follows:

- To find the shortest of a sequence of durations, the values must all be `xs:dayTimeDuration`s, or they must all be `xs:yearMonthDuration`s. You can't mix the two types of durations; if you do, the XSLT processor throws an error.
- If any of the items in the sequence are of type `xs:untypedAtomic`, the XSLT processor attempts to cast it to `xs:double`. If the item can't be converted to an `xs:double`, the XSLT processor throws an error.
- When comparing `xs:string` values, the default collation is used unless you specify another.
- Finally, if you pass the `min( )` function the empty sequence, it returns the empty sequence. Although you're not giving the function any useful data in this case, the XSLT processor doesn't throw an error.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.4, "Aggregate Functions."

Example

Here's a stylesheet that finds the minimum value in several different types of sequences:

```
<?xml version="1.0"?>
<!-- min.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="seq1" select="(3, 5, 18)"/>
    <xsl:variable name="seq2" select="(3, 5, 48.273, 2.9e3)"/>

    <xsl:variable name="value1" as="xs:integer" select="42"/>
    <xsl:variable name="value2" as="xs:double" select="2718.28E-3"/>
    <xsl:variable name="value3" as="xs:float" select="98.6"/>
    <xsl:variable name="value4" as="xs:decimal" select="2.54"/>
    <xsl:variable name="seq3"
      select="($value1, $value2, $value3, $value4)"/>

    <xsl:variable name="seq4"
      select="(xs:yearMonthDuration('P3Y8M'),
               xs:yearMonthDuration('P4Y2M'),
               xs:yearMonthDuration('P6Y4M'))"/>
    <xsl:variable name="seq5"
      select="(xs:dayTimeDuration('P2DT4H23M12.2S'),
               xs:dayTimeDuration('P3DT8H17M'),
               xs:dayTimeDuration('P3D'))"/>
    <xsl:variable name="seq6"
      select="('red', 'white', 'blue')"/>

    <xsl:text>&#xA;Here are some tests of the min() function:&#xA;</xsl:text>

    <xsl:text>&#xA;  min(</xsl:text>
    <xsl:value-of select="$seq1" separator=", "/>
    <xsl:text>) = </xsl:text>
    <xsl:value-of select="format-number(min($seq1), '#.###')"/>

    <xsl:text>&#xA;&#xA;  min(</xsl:text>
    <xsl:value-of select="$seq2" separator=", "/>
    <xsl:text>) = </xsl:text>
    <xsl:value-of select="format-number(min($seq2), '#.###')"/>

    <xsl:text>&#xA;&#xA;  min(</xsl:text>
    <xsl:value-of select="$seq3" separator=", "/>
    <xsl:text>) = </xsl:text>
    <xsl:value-of select="format-number(min($seq3), '#.###')"/>

    <xsl:text>&#xA;&#xA;  min(</xsl:text>
    <xsl:value-of select="$seq4" separator=", "/>
    <xsl:text>) = </xsl:text>
    <xsl:value-of select="min($seq4)"/>

    <xsl:text>&#xA;&#xA;    In text, the minimum of</xsl:text>
    <xsl:for-each select="$seq4">
      <xsl:text>&#xA;      </xsl:text>
```

```
    <xsl:value-of select="years-from-duration(.)"/>
    <xsl:text> years and </xsl:text>
    <xsl:value-of select="months-from-duration(.)"/>
    <xsl:text> months (</xsl:text>
    <xsl:value-of select="."/>
    <xsl:text>)</xsl:text>
  </xsl:for-each>

  <xsl:text>&#xA;    is </xsl:text>
  <xsl:value-of select="years-from-duration(min($seq4))"/>
  <xsl:text> years and </xsl:text>
  <xsl:value-of select="months-from-duration(min($seq4))"/>
  <xsl:text> months (</xsl:text>
  <xsl:value-of select="min($seq4)"/>
  <xsl:text>).</xsl:text>

  <xsl:text>&#xA;&#xA;  min(</xsl:text>
  <xsl:value-of select="$seq5" separator=", "/>
  <xsl:text>) = </xsl:text>
  <xsl:variable name="min5" select="min($seq5)"/>
  <xsl:value-of select="$min5"/>

  <xsl:text>&#xA;&#xA;    In text, the minimum of</xsl:text>
  <xsl:for-each select="$seq5">
    <xsl:text>&#xA;      </xsl:text>
    <xsl:value-of select="days-from-duration(.)"/>
    <xsl:text> days, </xsl:text>
    <xsl:value-of select="hours-from-duration(.)"/>
    <xsl:text> hours, </xsl:text>
    <xsl:value-of select="minutes-from-duration(.)"/>
    <xsl:text> minutes and </xsl:text>
    <xsl:value-of
      select="format-number(seconds-from-duration(.), '#.##')"/>
    <xsl:text> seconds (</xsl:text>
    <xsl:value-of select="."/>
    <xsl:text>)</xsl:text>
  </xsl:for-each>

  <xsl:text>&#xA;    is </xsl:text>
  <xsl:value-of select="days-from-duration($min5)"/>
  <xsl:text> days, </xsl:text>
  <xsl:value-of select="hours-from-duration($min5)"/>
  <xsl:text> hours, </xsl:text>
  <xsl:value-of select="minutes-from-duration($min5)"/>
  <xsl:text> minutes and </xsl:text>
  <xsl:value-of
    select="format-number(seconds-from-duration($min5), '#.##')"/>
  <xsl:text> seconds.</xsl:text>

  <xsl:text>&#xA;&#xA;  min(</xsl:text>
  <xsl:value-of select="$seq6" separator=", "/>
  <xsl:text>) = </xsl:text>
  <xsl:value-of select="min($seq6)"/>

</xsl:template>
```

```
</xsl:stylesheet>
```

Here are the results from this stylesheet:

```
Here are some tests of the min() function:

  min(3, 5, 18) = 3

  min(3, 5, 48.273, 2900) = 3

  min(42, 2.71828, 98.6, 2.54) = 2.54

  min(P3Y8M, P4Y2M, P6Y4M) = P3Y8M

    In text, the minimum of
      3 years and 8 months (P3Y8M)
      4 years and 2 months (P4Y2M)
      6 years and 4 months (P6Y4M)
    is 3 years and 8 months (P3Y8M).

  min(P2DT4H23M12.2S, P3DT8H17M, P3D) = P2DT4H23M12.2S

    In text, the minimum of
      2 days, 4 hours, 23 minutes and 12.2 seconds (P2DT4H23M12.2S)
      3 days, 8 hours, 17 minutes and 0 seconds (P3DT8H17M)
      3 days, 0 hours, 0 minutes and 0 seconds (P3D)
    is 2 days, 4 hours, 23 minutes and 12.2 seconds.
```

The stylesheet demonstrates six different types of sequences. The first is all integers and the second is a sequence of numbers. All the values in the first two sequences can be converted to numbers, so the results are what we expect. The third sequence features four numeric values, each of which has a specific numeric type. The next two sequences are made up of the two types of durations, and the final sequence is made up of strings. The stylesheet also uses functions such as `years-from-duration( )` and `format-number( )` to format the data.

Here's a short example that illustrates using a custom collation:

```
<?xml version="1.0"?>
<!-- min2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="seq1"
      select="('stra&#xDF;e', 'strasse')"/>
    <xsl:variable name="seq2" select="reverse($seq1)"/>

    <xsl:text>&#xA;Here are some tests of the min() function:&#xA;</xsl:text>

    <xsl:text>&#xA;  min(</xsl:text>
    <xsl:value-of select="$seq1" separator=", "/>
    <xsl:text>) = </xsl:text>
```

```
    <xsl:value-of select="min($seq1)"/>

    <xsl:text>&#xA;  min(</xsl:text>
    <xsl:value-of select="$seq1" separator=", "/>
    <xsl:text> [German collation]) = </xsl:text>
    <xsl:value-of
      select="min($seq1,
              concat('http://saxon.sf.net/collation?',
                     'class=com.oreilly.xslt.GermanCollation;'))"/>

    <xsl:text>&#xA;  min(</xsl:text>
    <xsl:value-of select="$seq2" separator=", "/>
    <xsl:text> [German collation]) = </xsl:text>
    <xsl:value-of
      select="min($seq2,
              concat('http://saxon.sf.net/collation?',
                     'class=com.oreilly.xslt.GermanCollation;'))"/>

  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Here are some tests of the min() function:

  min(strasse, straße) = strasse
  min(straße, strasse [German collation]) = straße
  min(strasse, straße [German collation]) = strasse
```

Using the default collation, these two strings are not equal. In the first example, `strasse` sorts before `straße`, so that's the value returned by the `min( )` function. In the final two examples, the two strings are considered equal by the German collation. In Saxon 9.0.0.3J, `min( )` returns the minimum value that appears first in the sequence. This is implementation dependent; XSLT processors are allowed to return any of the equal minimum values.

[2.0] minutes-from-dateTime()

Given an `xs:dateTime`, returns its minutes value.

Syntax

```
xs:integer? minutes-from-dateTime(xs:dateTime?)
```

Inputs

An `xs:dateTime` value.

Output

An `xs:integer` representing the minutes component of the given `xs:dateTime` value. If the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `minutes-from-dateTime( )` function:

```
<?xml version="1.0"?>
<!-- minutes-from-datetime.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the minutes from an xs:dateTime:</xsl:text>
    <xsl:variable name="currentDateTime" as="xs:dateTime"
      select="current-dateTime()"/>
    <xsl:text>&#xA;&#xA;  The current date and time is: </xsl:text>
    <xsl:value-of select="$currentDateTime"/>

    <xsl:text>&#xA;&#xA;  The current minute is: </xsl:text>
    <xsl:value-of select="minutes-from-dateTime($currentDateTime)"/>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet creates these results:

```
Extracting the minutes from an xs:dateTime:

  The current date and time is: 2006-11-16T04:34:19.921-05:00

  The current minute is: 34
```

See Also

The definitions of the [2.0] `day-from-dateTime( )`, [2.0] `format-dateTime( )`, [2.0] `hours-from-dateTime( )`, [2.0] `month-from-dateTime( )`, [2.0] `seconds-from-dateTime( )`, [2.0] `timezone-from-dateTime( )`, and [2.0] `year-from-dateTime( )` functions.

[2.0] minutes-from-duration()

Given an `xs:duration` value, returns the number of minutes in that duration.

Syntax

```
xs:integer? minutes-from-duration(xs:duration?)
```

Inputs

An `xs:duration` value.

Output

An `xs:integer` representing the minutes component of the given `xs:duration`. Be aware that for an `xs:yearMonthDuration`, this function always returns `0` because there is no minutes component of an `xs:yearMonthDuration`. Also, if the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `minutes-from-duration( )` function with all three types of durations:

```
<?xml version="1.0"?>
<!-- minutes-from-duration.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the minutes component from durations:</xsl:text>

    <xsl:variable name="sampleDuration" as="xs:duration"
      select="xs:duration('P3Y8M2DT4H23M12.2S')"/>
    <xsl:variable name="sampleYearMonthDuration" as="xs:yearMonthDuration"
      select="xs:yearMonthDuration('P3Y8M')"/>
    <xsl:variable name="sampleDayTimeDuration" as="xs:dayTimeDuration"
      select="xs:dayTimeDuration('P2DT4H23M12.2S')"/>

    <xsl:text>&#xA;&#xA;  A sample xs:duration: </xsl:text>
    <xsl:value-of select="$sampleDuration"/>
    <xsl:text>&#xA;    The minutes component of this duration is </xsl:text>
    <xsl:value-of select="minutes-from-duration($sampleDuration)"/>
    <xsl:text>.</xsl:text>

    <xsl:text>&#xA;&#xA;  A sample xs:yearMonthDuration: </xsl:text>
    <xsl:value-of select="$sampleYearMonthDuration"/>
    <xsl:text>&#xA;    The minutes component of this duration is </xsl:text>
    <xsl:value-of select="minutes-from-duration($sampleYearMonthDuration)"/>
    <xsl:text>.</xsl:text>

    <xsl:text>&#xA;&#xA;  A sample xs:dayTimeDuration: </xsl:text>
    <xsl:value-of select="$sampleDayTimeDuration"/>
    <xsl:text>&#xA;    The minutes component of this duration is </xsl:text>
    <xsl:value-of select="minutes-from-duration($sampleDayTimeDuration)"/>
    <xsl:text>.</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates these results:

```
Extracting the minutes component from durations:

  A sample xs:duration: P3Y8M2DT4H23M12.2S
    The minutes component of this duration is 23.

  A sample xs:yearMonthDuration: P3Y8M
    The minutes component of this duration is 0.

  A sample xs:dayTimeDuration: P2DT4H23M12.2S
    The minutes component of this duration is 23.
```

Extracting the minutes component from an `xs:yearMonthDuration` returns `0`, as you'd expect.

See Also

The definitions of the [2.0] `days-from-duration( )`, [2.0] `hours-from-duration( )`, [2.0] `months-from-duration( )`, [2.0] `seconds-from-duration( )`, and [2.0] `years-from-duration( )` functions.

[2.0] minutes-from-time()

Given an `xs:time` value, returns its minutes component.

Syntax

```
xs:integer? minutes-from-time(xs:time?)
```

Input

An `xs:time` value.

Output

An `xs:integer` representing the minutes component of the given `xs:time` value. If the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

Here's a stylesheet that uses `minutes-from-time( )`:

```
<?xml version="1.0"?>
<!-- minutes-from-time.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>
```

```
  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the minutes component from an xs:time:</xsl:text>
    <xsl:variable name="currentTime" as="xs:time" select="current-time()"/>
    <xsl:text>&#xA;&#xA;  The current time is: </xsl:text>
    <xsl:value-of select="$currentTime"/>

    <xsl:text>&#xA;&#xA;  The current minute is: </xsl:text>
    <xsl:value-of select="minutes-from-time($currentTime)"/>
    <xsl:text>, the </xsl:text>
    <xsl:value-of select="format-time($currentTime, '[mwo]')"/>
    <xsl:text> minute of the hour.</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet creates these results:

```
Extracting the minutes component from an xs:time:

  The current time is: 04:39:49.875-05:00

  The current minute is: 39, the thirty-ninth minute of the hour.
```

Notice that the first result was generated by the `minutes-from-time( )` function, while the second result was generated by using `format-time( )` with a format string that selected the minutes component from the `xs:time` value.

See Also

The definitions of the [2.0] `format-time( )`, [2.0] `hours-from-time( )`, [2.0] `seconds-from-time( )`, and [2.0] `timezone-from-time( )`.

[2.0] month-from-date()

Given an `xs:date` value, returns its month value.

Syntax

```
xs:integer? month-from-date(xs:date?)
```

Input

An `xs:date` value.

Output

An `xs:integer` representing the month component of the given `xs:date` value. If the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `month-from-date( )` function:

```
<?xml version="1.0"?>
<!-- month-from-date.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the month from an xs:date:</xsl:text>
    <xsl:variable name="currentDate" as="xs:date" select="current-date()"/>
    <xsl:text>&#xA;&#xA;The current date is: </xsl:text>
    <xsl:value-of select="$currentDate"/>

    <xsl:text>&#xA;&#xA;  The current month: </xsl:text>
    <xsl:value-of select="month-from-date($currentDate)"/>
    <xsl:text>&#xA;    In English: </xsl:text>
    <xsl:value-of select="format-date($currentDate, '[MNn]')"/>
    <xsl:text>&#xA;    In German: </xsl:text>
    <xsl:value-of select="format-date($currentDate, '[MNn]', 'de', (), ())"/>
    <xsl:text>&#xA;    It's the </xsl:text>
    <xsl:value-of select="format-date($currentDate, '[M1o]')"/>
    <xsl:text> month of the year.</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet creates these results:

```
Extracting the month from an xs:dateTime:

The current date and time is: 2006-07-05T12:38:00.877-05:00

  The current month: 7
    In English: July
    In German: Juli
    It's the 7th month of the year.
```

Notice that the first result was generated by the `month-from-date( )` function, while the other results were generated by using `format-date( )` with a format string that selected only the month component of the `xs:date` value.

See Also

The definitions of the [2.0] `day-from-date( )`, [2.0] `format-date( )`, [2.0] `timezone-from-date( )`, and [2.0] `year-from-date( )` functions.

[2.0] month-from-dateTime()

Given an `xs:dateTime` value, returns its month value.

Syntax

```
xs:integer? month-from-dateTime(xs:dateTime?)
```

Inputs

An `xs:dateTime` value.

Output

An `xs:integer` representing the month component of the given `xs:dateTime` value. If the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `month-from-dateTime( )` function:

```
<?xml version="1.0"?>
<!-- month-from-datetime.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the month from an xs:dateTime:</xsl:text>
    <xsl:variable name="currentDateTime" as="xs:dateTime"
      select="current-dateTime()"/>
    <xsl:text>&#xA;&#xA;The current date and time is: </xsl:text>
    <xsl:value-of select="$currentDateTime"/>

    <xsl:text>&#xA;&#xA;  The current month: </xsl:text>
    <xsl:value-of select="month-from-dateTime($currentDateTime)"/>
    <xsl:text>&#xA;    In English: </xsl:text>
    <xsl:value-of select="format-dateTime($currentDateTime, '[MNn]')"/>
    <xsl:text>&#xA;    In German: </xsl:text>
    <xsl:value-of select="format-dateTime($currentDateTime, '[MNn]', 'de', (), ())"/>
    <xsl:text>&#xA;    It's the </xsl:text>
    <xsl:value-of select="format-dateTime($currentDateTime, '[M1o]')"/>
    <xsl:text> month of the year.</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet creates these results:

```
Extracting the month from an xs:dateTime:

The current date and time is: 2006-11-16T04:42:47.481-05:00

  The current month: 11
    In English: November
    In German: November
    It's the 11th month of the year.
```

Notice that some of the results were generated by the `month-from-dateTime( )` function, while other results were generated by using `format-dateTime( )` with a format string that selected only the month component of the `xs:dateTime` value.

See Also

The definitions of the [2.0] `day-from-dateTime( )`, [2.0] `format-dateTime( )`, [2.0] `hours-from-dateTime( )`, [2.0] `minutes-from-dateTime( )`, [2.0] `seconds-from-dateTime( )`, [2.0] `timezone-from-dateTime( )`, and [2.0] `year-from-dateTime( )` functions.

[2.0] months-from-duration()

Given an `xs:duration` value, returns the number of months in that duration.

Syntax

```
xs:integer? months-from-duration(xs:duration?)
```

Inputs

An `xs:duration` value.

Output

An `xs:integer` representing the months component of the given `xs:duration`. Be aware that for an `xs:dayTimeDuration`, this function always returns `0` because there is no months component of an `xs:dayTimeDuration`. Also, if the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `months-from-duration( )` function with all three types of durations:

```
<?xml version="1.0"?>
<!-- months-from-duration.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```
<xsl:output method="text"/>

<xsl:template match="/">
  <xsl:text>&#xA;Extracting the months component from durations:</xsl:text>

  <xsl:variable name="sampleDuration" as="xs:duration"
    select="xs:duration('P3Y8M2DT4H23M12.2S')"/>
  <xsl:variable name="sampleYearMonthDuration" as="xs:yearMonthDuration"
    select="xs:yearMonthDuration('P3Y8M')"/>
  <xsl:variable name="sampleDayTimeDuration" as="xs:dayTimeDuration"
    select="xs:dayTimeDuration('P2DT4H23M12.2S')"/>

  <xsl:text>&#xA;&#xA;  A sample xs:duration: </xsl:text>
  <xsl:value-of select="$sampleDuration"/>
  <xsl:text>&#xA;    The months component of this duration is </xsl:text>
  <xsl:value-of select="months-from-duration($sampleDuration)"/>
  <xsl:text>.</xsl:text>

  <xsl:text>&#xA;&#xA;  A sample xs:yearMonthDuration: </xsl:text>
  <xsl:value-of select="$sampleYearMonthDuration"/>
  <xsl:text>&#xA;    The months component of this duration is </xsl:text>
  <xsl:value-of select="months-from-duration($sampleYearMonthDuration)"/>
  <xsl:text>.</xsl:text>

  <xsl:text>&#xA;&#xA;  A sample xs:dayTimeDuration: </xsl:text>
  <xsl:value-of select="$sampleDayTimeDuration"/>
  <xsl:text>&#xA;    The months component of this duration is </xsl:text>
  <xsl:value-of select="months-from-duration($sampleDayTimeDuration)"/>
  <xsl:text>.</xsl:text>
</xsl:template>

</xsl:stylesheet>
```

This stylesheet generates these results:

```
Extracting the months component from durations:

  A sample xs:duration: P3Y8M2DT4H23M12.2S
    The months component of this duration is 8.

  A sample xs:yearMonthDuration: P3Y8M
    The months component of this duration is 8.

  A sample xs:dayTimeDuration: P2DT4H23M12.2S
    The months component of this duration is 0.
```

As you can see from the results, extracting the months component from an `xs:dayTimeDuration` returns `0`.

See Also

The definitions of the [2.0] `days-from-duration( )`, [2.0] `hours-from-duration( )`, [2.0] `minutes-from-duration( )`, [2.0] `seconds-from-duration( )`, and the [2.0] `years-from-duration( )` functions.

name()

Returns the qualified name of a node. The qualified name includes the appropriate namespace prefix. For information on the namespace URI (not the prefix), XPath provides the `namespace-uri( )` function.

Syntax

[1.0] `string` **`name(`***`node-set?`***`)`**
[2.0] `xs:string` **`name(`***`node()?`***`)`**

Inputs

[1.0] An optional node-set. If the node-set is present, the XSLT 1.0 processor invokes the `name( )` function against the first node in the node-set. If no node-set is given, the `name( )` function applies to the context node.

[2.0] An optional node sequence. *In XSLT 2.0, it is an error to call the* `name( )` *function with a sequence containing more than one node.* If the node sequence is not present, the `name( )` function applies to the context node.

Output

The qualified name of the node. If the argument to the `name( )` function is empty (say we use `select="item"` where there are no `<item>` elements, for example), `name( )` returns an empty string.

Defined in

[1.0] XPath section 4.1, "Node Set Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 14, "Functions and Operators on Nodes."

Example

Here is the XML document we'll use to demonstrate the `name( )` function:

```
<?xml version="1.0"?>
<!-- sonnet.xml -->
<sonnet type='Shakespearean'>
  <auth:author xmlns:auth="http://www.authors.com/">
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
    <year-of-death>1616</year-of-death>
  </auth:author>
  <!-- Is there an official title for this sonnet?  They're
       sometimes named after the first line.                    -->
  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
```

```
    <line>If hairs be wires, black wires grow on her head.</line>
    <line>I have seen roses damasked, red and white,</line>
    <line>But no such roses see I in her cheeks.</line>
    <line>And in some perfumes is there more delight</line>
    <line>Than in the breath that from my mistress reeks.</line>
    <line>I love to hear her speak, yet well I know</line>
    <line>That music hath a far more pleasing sound.</line>
    <line>I grant I never saw a goddess go,</line>
    <line>My mistress when she walks, treads on the ground.</line>
    <line>And yet, by Heaven, I think my love as rare</line>
    <line>As any she belied with false compare.</line>
  </lines>
</sonnet>
```

We'll use this stylesheet to output the value of the `name( )` function for each node in the XML document:

```
<?xml version="1.0"?>
<!-- name.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the name() function:&#xA;</xsl:text>

    <xsl:for-each select="//*">
      <xsl:text>&#xA;  Element </xsl:text>
      <xsl:number level="any" count="*"/>
      <xsl:text>: </xsl:text>
      <xsl:value-of select="name()"/>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

When we transform the XML document with this stylesheet, here are the results:

```
A test of the name() function:

  Element 1: sonnet
  Element 2: auth:author
  Element 3: last-name
  Element 4: first-name
  Element 5: nationality
  Element 6: year-of-birth
  Element 7: year-of-death
  Element 8: title
  Element 9: lines
  Element 10: line
  Element 11: line
  Element 12: line
  Element 13: line
  Element 14: line
  Element 15: line
  Element 16: line
```

```
  Element 17: line
  Element 18: line
  Element 19: line
  Element 20: line
  Element 21: line
  Element 22: line
  Element 23: line
```

Here's a slightly more advanced version in which we use the XSLT 2.0 `<xsl:sequence>` element to select a sequence of nodes from the document. We store that sequence in a variable, and then iterate through the values of the variable and print the names of the elements:

```
<?xml version="1.0"?>
<!-- name2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.oreilly.com/xslt"
  xmlns:auth="http://www.authors.com/">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the name() function:&#xA;</xsl:text>

    <xsl:variable name="authorDetails" as="element()*"
      select="/sonnet/auth:author/*"/>
    <xsl:for-each select="$authorDetails">
      <xsl:text>&#xA;  Element </xsl:text>
      <xsl:value-of select="position()"/>
      <xsl:text>: </xsl:text>
      <xsl:value-of select="name()"/>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

(Notice that the sequence is stored in a variable with type `element()*`, and that we had to define the `auth` namespace in the stylesheet.) The results contain the names of all the elements that appear below the `<auth:author>` element in the source document:

```
Tests of the name() function:

  Element 1: last-name
  Element 2: first-name
  Element 3: nationality
  Element 4: year-of-birth
  Element 5: year-of-death
```

As we mentioned earlier, it is an error in XSLT 2.0 to pass more than one node to the `name()` function. (In XSLT 1.0, the processor simply used the first node in the node-set and ignored everything else.) It is possible to pass a variable with type `element()*` to the `name()` function, but that sequence cannot contain more than one member. For example, the following is legal in XSLT 2.0:

```
<xsl:variable name="salesFigures" as="element()*"
  select="/report/brand[1]/*[2]"/>
<xsl:value-of select="name($salesFigures)"/>
```

This is legal because the XPath expression in the `<xsl:variable>` element selects only a single node. This code, however, causes the XSLT 2.0 processor to raise an error:

```
<xsl:variable name="salesFigures" as="element()*"
  select="/report/brand[1]/*"/>
<xsl:value-of select="name($salesFigures)"/>
```

This is an error because the XPath expression selects more than one node from our sample document. If we change the XML source document so that there is only one node under the first `<brand>` element, the stylesheet would work.

namespace-uri()

Returns the namespace URI of the argument node.

Syntax

[1.0] string **namespace-uri(***node-set?***)**
[2.0] xs:anyURI **namespace-uri()**
[2.0] xs:anyURI **namespace-uri(***node()?***)**

Inputs

[1.0] A node-set. If the node-set is omitted, the `namespace-uri( )` function creates a node-set that has the context node as its only member.

[2.0] A node. If the node is omitted, the `namespace-uri( )` function is evaluated against the context item. If the input is a sequence with more than one node, the XSLT processor raises an error.

Output

[1.0] The namespace URI of the first node in the argument node-set. If the argument node-set is empty, the first node has no namespace URI, or the first node has a namespace URI that is null, an empty string is returned.

[2.0] The namespace URI of the argument node. If the argument is the empty sequence, the argument has no namespace URI or the node has a namespace URI that is null, an empty string is returned.

Be aware that the `namespace-uri( )` function returns an empty string for all nodes other than element and attribute nodes.

Defined in

[1.0] XPath section 4.1, “Node Set Functions.”

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 14, “Functions and Operators on Nodes.”

Example

We'll use a slightly modified version of our Shakespearean sonnet to illustrate the `namespace-uri( )` function:

```
<?xml version="1.0"?>
<!-- sonnet-namespace.xml -->
<sonnet type='Shakespearean'
  xmlns="http://www.oreilly.com/xslt">
  <author xmlns="http://www.authors.com/">
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
    <year-of-death>1616</year-of-death>
  </author>
  <!-- Is there an official title for this sonnet?  They're
       sometimes named after the first line.                    -->
  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
    <line>If hairs be wires, black wires grow on her head.</line>
    <line>I have seen roses damasked, red and white,</line>
    <line>But no such roses see I in her cheeks.</line>
    <line>And in some perfumes is there more delight</line>
    <line>Than in the breath that from my mistress reeks.</line>
    <line>I love to hear her speak, yet well I know</line>
    <line>That music hath a far more pleasing sound.</line>
    <line>I grant I never saw a goddess go,</line>
    <line>My mistress when she walks, treads on the ground.</line>
    <line>And yet, by Heaven, I think my love as rare</line>
    <line>As any she belied with false compare.</line>
  </lines>
</sonnet>
```

The sonnet has been modified so that it has a default namespace (`http://www.oreilly.com/`). Our sample stylesheet goes through any XML document and invokes `namespace-uri( )` against every element and attribute node in the document. Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- namespace-uri.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the namespace-uri() function:&#xA;&#xA;</xsl:text>

    <xsl:for-each select="//*">
      <xsl:text>  Namespace URI for &lt;</xsl:text>
      <xsl:value-of select="node-name(.)"/>
      <xsl:text>&gt;: </xsl:text>
      <xsl:value-of select="namespace-uri()"/>
```

```
      <xsl:text>&#xA;</xsl:text>
      <xsl:for-each select="@*">
        <xsl:text>    Namespace URI for attribute "</xsl:text>
        <xsl:value-of select="node-name(.)"/>
        <xsl:text>": </xsl:text>
        <xsl:value-of select="namespace-uri()"/>
        <xsl:text>&#xA;</xsl:text>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results of our stylesheet:

```
A test of the namespace-uri() function:

  Namespace URI for <sonnet>: http://www.oreilly.com/xslt
    Namespace URI for attribute "type":
  Namespace URI for <author>: http://www.authors.com/
  Namespace URI for <last-name>: http://www.authors.com/
  Namespace URI for <first-name>: http://www.authors.com/
  Namespace URI for <nationality>: http://www.authors.com/
  Namespace URI for <year-of-birth>: http://www.authors.com/
  Namespace URI for <year-of-death>: http://www.authors.com/
  Namespace URI for <title>: http://www.oreilly.com/xslt
  Namespace URI for <lines>: http://www.oreilly.com/xslt
  Namespace URI for <line>: http://www.oreilly.com/xslt
  Namespace URI for <line>: http://www.oreilly.com/xslt
  Namespace URI for <line>: http://www.oreilly.com/xslt
  Namespace URI for <line>: http://www.oreilly.com/xslt
  Namespace URI for <line>: http://www.oreilly.com/xslt
  Namespace URI for <line>: http://www.oreilly.com/xslt
  Namespace URI for <line>: http://www.oreilly.com/xslt
  Namespace URI for <line>: http://www.oreilly.com/xslt
  Namespace URI for <line>: http://www.oreilly.com/xslt
  Namespace URI for <line>: http://www.oreilly.com/xslt
  Namespace URI for <line>: http://www.oreilly.com/xslt
  Namespace URI for <line>: http://www.oreilly.com/xslt
  Namespace URI for <line>: http://www.oreilly.com/xslt
```

In our modified sonnet, the `<author>` element declares a new default namespace, so all of the elements inside it return the default namespace. Outside the `<author>` element, the original default namespace comes back into scope.

[2.0] namespace-uri-for-prefix()

Given a namespace prefix and an element, returns the URI associated with that prefix.

Syntax

```
xs:anyURI? namespace-uri-for-prefix($prefix as xs:string?,
                                    $element as element())
```

Inputs

A namespace prefix and an element. If the prefix is an empty string, the URI for the default namespace is returned, assuming a default namespace is in scope.

Output

The URI associated with the namespace prefix for the given element.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 11.2, "Operators and Functions Related to QNames."

Example

For an example, we'll reuse our Shakespearean sonnet:

```
<?xml version="1.0"?>
<!-- sonnet.xml -->
<sonnet type='Shakespearean'>
  <auth:author xmlns:auth="http://www.authors.com/">
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
    <year-of-death>1616</year-of-death>
  </auth:author>
  <!-- Is there an official title for this sonnet?  They're
       sometimes named after the first line.                    -->
  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
    <line>If hairs be wires, black wires grow on her head.</line>
    <line>I have seen roses damasked, red and white,</line>
    <line>But no such roses see I in her cheeks.</line>
    <line>And in some perfumes is there more delight</line>
    <line>Than in the breath that from my mistress reeks.</line>
    <line>I love to hear her speak, yet well I know</line>
    <line>That music hath a far more pleasing sound.</line>
    <line>I grant I never saw a goddess go,</line>
    <line>My mistress when she walks, treads on the ground.</line>
    <line>And yet, by Heaven, I think my love as rare</line>
    <line>As any she belied with false compare.</line>
  </lines>
</sonnet>
```

And here's our stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- namespace-uri-for-prefix.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>
```

```
  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the namespace-uri-for-prefix() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>&#xA;  namespace-uri-for-prefix</xsl:text>
    <xsl:text>('auth', //last-name) = "</xsl:text>
    <xsl:value-of
      select="namespace-uri-for-prefix('auth', //last-name)"/>
    <xsl:text>"</xsl:text>

    <xsl:text>&#xA;  namespace-uri-for-prefix</xsl:text>
    <xsl:text>('auth', //*:author) = "</xsl:text>
    <xsl:value-of
      select="namespace-uri-for-prefix('auth', //*:author)"/>
    <xsl:text>"</xsl:text>

    <xsl:text>&#xA;  namespace-uri-for-prefix</xsl:text>
    <xsl:text>('auth', /sonnet) = "</xsl:text>
    <xsl:value-of
      select="namespace-uri-for-prefix('auth', /sonnet)"/>
    <xsl:text>"</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Tests of the namespace-uri-for-prefix() function:

  namespace-uri-for-prefix('auth', //last-name) = "http://www.authors.com/"
  namespace-uri-for-prefix('auth', //*:author) = "http://www.authors.com/"
  namespace-uri-for-prefix('auth', /sonnet) = ""
```

In our stylesheet, we're looking for the URI for the prefix `auth`. In the first test of `namespace-uri-for-prefix( )`, the `auth` prefix is in scope for the `<auth:author>` element and everything inside it, so XSLT returns the URL associated with it. For the second test, we choose to use a wildcard in the XPath expression. (The function call `namespace-uri-for-prefix('auth', //auth:author)` requires us to define the `auth` namespace in our stylesheet.) Finally, the last example returns an empty string because the `auth` prefix is not in scope for the `<sonnet>` element.

This stylesheet was written with knowledge of the XML document. A more complicated example would use a function such as `in-scope-prefixes( )` to retrieve a sequence of prefixes, and then use `namespace-uri-for-prefix( )` to retrieve the URI associated with each prefix.

There are other complications: It is a fatal error to pass the empty sequence or a sequence of more than one item as the second argument. That's not a problem here because we wrote our stylesheet knowing there were `<last-name>`, `<auth:author>`, and `<sonnet>` elements in the document, and that each of them occurred once.

As always, feel free to implement a more robust stylesheet for your own amusement.

[2.0] namespace-uri-from-QName()

Given an `xs:QName` value, returns the namespace URI associated with that `QName`.

Syntax

```
xs:anyURI? namespace-uri-from-QName(xs:QName?)
```

Input

An `xs:QName` value.

Output

The namespace URI associated with this `QName`. If the argument is the empty sequence, the empty sequence is returned. If the `QName` is not in a namespace, a zero-length string is returned.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 11.2, "Operators and Functions Related to QNames."

Example

We'll reuse our Shakespearean sonnet as our XML input document here:

```
<?xml version="1.0"?>
<!-- sonnet.xml -->
<sonnet type='Shakespearean'>
  <auth:author xmlns:auth="http://www.authors.com/">
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
    <year-of-death>1616</year-of-death>
  </auth:author>
  <!-- Is there an official title for this sonnet?  They're
       sometimes named after the first line.                    -->
  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
    <line>If hairs be wires, black wires grow on her head.</line>
    <line>I have seen roses damasked, red and white,</line>
    <line>But no such roses see I in her cheeks.</line>
    <line>And in some perfumes is there more delight</line>
    <line>Than in the breath that from my mistress reeks.</line>
    <line>I love to hear her speak, yet well I know</line>
    <line>That music hath a far more pleasing sound.</line>
    <line>I grant I never saw a goddess go,</line>
    <line>My mistress when she walks, treads on the ground.</line>
    <line>And yet, by Heaven, I think my love as rare</line>
    <line>As any she belied with false compare.</line>
  </lines>
</sonnet>
```

Here is our stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- namespace-uri-from-QName.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the namespace-uri-from-QName() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:variable name="testQName1" as="xs:QName"
      select="resolve-QName('auth:last-name',
              /sonnet/*:author/last-name)"/>
    <xsl:text>&#xA;  testQName1 = resolve-QName('auth:last-name', </xsl:text>
    <xsl:text>/sonnet/*:author/last-name)</xsl:text>
    <xsl:text>&#xA;  namespace-uri-from-QName($testQName1) = "</xsl:text>
    <xsl:value-of select="namespace-uri-from-QName($testQName1)"/>
    <xsl:text>"</xsl:text>

    <xsl:variable name="testQName2" as="xs:QName"
      select="resolve-QName('something-else', /sonnet)"/>
    <xsl:text>&#xA;&#xA;  testQName2 = resolve-QName('</xsl:text>
    <xsl:text>something-else', /sonnet)</xsl:text>
    <xsl:text>&#xA;  namespace-uri-from-QName($testQName2) = "</xsl:text>
    <xsl:value-of select="namespace-uri-from-QName($testQName2)"/>
    <xsl:text>"</xsl:text>

    <xsl:variable name="testQName3" as="xs:QName"
      select="QName('http://www.authors.com/', 'pfx:writer')"/>
    <xsl:text>&#xA;&#xA;  testQName3 = QName('http://www.</xsl:text>
    <xsl:text>authors.com/', 'pfx:writer')&#xA;</xsl:text>
    <xsl:text>  namespace-uri-from-QName($testQName3) = "</xsl:text>
    <xsl:value-of select="namespace-uri-from-QName($testQName3)"/>
    <xsl:text>"</xsl:text>

  </xsl:template>

</xsl:stylesheet>
```

Here are our results:

```
Tests of the namespace-uri-from-QName() function:

  testQName1 = resolve-QName('auth:last-name', /sonnet/*:author/last-name)
  namespace-uri-from-QName($testQName1) = "http://www.authors.com/"

  testQName2 = resolve-QName('something-else', /sonnet)
  namespace-uri-from-QName($testQName2) = ""

  testQName3 = QName('http://www.authors.com/', 'pfx:writer')
  namespace-uri-from-QName($testQName3) = "http://www.authors.com/"
```

The argument to the namespace-uri-from-QName( ) function *must* be an xs:QName. In this stylesheet, we create QNames with the resolve-QName( ) function and the QName( ) constructor function.

For the first test, we create a QName with a qualified element name of auth:last-name. The element (the second argument to resolve-QName( )) is the <last-name> element in our sonnet. The resolve-QName( ) function returns a QName with a local name of last-name, a namespace prefix of auth, and a namespace URI of http://www.authors.com/. If there are no nodes that match the XPath expression /sonnet/*:author/last-name, the call to resolve-QName( ) fails. The call also fails if the auth prefix is not declared for the <last-name> element.

The second test of namespace-uri-from-QName( ) uses an unprefixed element name in the call to resolve-QName( ). The XML document doesn't have a default namespace, so our QName has an empty string for its namespace prefix and namespace URI.

For the final test, we use the QName( ) function to create a QName from scratch. The two arguments to the function are the namespace URI and an element name—in this case, a prefixed element name. The resulting QName has a local name of writer, a namespace prefix of pfx, and a namespace URI of http://www.authors.com/.

We'll look at a couple more examples, but this time we'll use a version of the sonnet that *does* have a default namespace:

```
<?xml version="1.0"?>
<!-- sonnet-default-namespace.xml -->
<sonnet type='Shakespearean'
  xmlns="http://www.oreilly.com/xslt">
  <auth:author xmlns:auth="http://www.authors.com/">
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
...
</sonnet>
```

The default namespace in the new version of our sonnet is http://www.oreilly.com/xslt. Now we need to modify our stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- namespace-uri-from-QName2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the namespace-uri-from-QName() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:variable name="testQName1" as="xs:QName"
      select="resolve-QName('auth:last-name',
              /*:sonnet/*:author/*:last-name)"/>
    <xsl:text>&#xA;  testQName1 = resolve-QName('auth:last-name', </xsl:text>
    <xsl:text>/*:sonnet/*:author/*:last-name)</xsl:text>
```

```
    <xsl:text>&#xA;  namespace-uri-from-QName($testQName1) = "</xsl:text>
    <xsl:value-of select="namespace-uri-from-QName($testQName1)"/>
    <xsl:text>"</xsl:text>

    <xsl:variable name="testQName2" as="xs:QName"
      select="resolve-QName('something-else', /*:sonnet)"/>
    <xsl:text>&#xA;&#xA;  testQName2 = resolve-QName('</xsl:text>
    <xsl:text>something-else', /*:sonnet)</xsl:text>
    <xsl:text>&#xA;  namespace-uri-from-QName($testQName2) = "</xsl:text>
    <xsl:value-of select="namespace-uri-from-QName($testQName2)"/>
    <xsl:text>"</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

Now we get slightly different results:

```
Tests of the namespace-uri-from-QName() function:

  testQName1 = resolve-QName('auth:last-name', /*:sonnet/*:author/*:last-name)
  namespace-uri-from-QName($testQName1) = "http://www.authors.com/"

  testQName2 = resolve-QName('something-else', /*:sonnet)
  namespace-uri-from-QName($testQName2) = "http://www.oreilly.com/xslt"
```

The difference in these results is that we get a namespace URI for the `QName` we created without a prefix. Our modified sonnet has a default namespace, so we get the value of that namespace.

The change in our stylesheet is that we had to use asterisks to refer to items in the `http://www.oreilly.com/xslt` namespace. In other words, we had to replace `/sonnet/*:auth/last-name` with `/*:sonnet/*:auth/*:last-name`. One way of referring to the default namespace from the XML document would be to declare it with a namespace prefix in our stylesheet.

Another approach is that we could give the sonnet's default namespace a prefix in our stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- namespace-uri-from-QName3.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:ora="http://www.oreilly.com/xslt">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the namespace-uri-from-QName() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:variable name="testQName1" as="xs:QName"
      select="resolve-QName('auth:last-name',
              /ora:sonnet/*:author/ora:last-name)"/>
  ...
  </xsl:template>

</xsl:stylesheet>
```

This gives us results similar to those in our second stylesheet:

```
Tests of the namespace-uri-from-QName() function:

  testQName1 = resolve-QName('auth:last-name',
                             /ora:sonnet/*:author/ora:last-name)
  namespace-uri-from-QName($testQName1) = "http://www.authors.com/"

  testQName2 = resolve-QName('something-else', /ora:sonnet)
  namespace-uri-from-QName($testQName2) = "http://www.oreilly.com/xslt"
```

We associated the `ora` namespace prefix with the default namespace in the XML source document. Whenever we use the `ora` prefix in our stylesheet, it refers to the `http://www.oreilly.com/xslt` namespace. That's the default namespace of the XML document we're processing, so we can use a specific namespace instead of a wildcard.

[2.0 – Schema] nilled()

Returns `true` if a given element has been nilled, `false` otherwise. If the argument is not an element, `nilled( )` returns the empty sequence.

Syntax

```
xs:boolean? nilled(node()?)
```

Inputs

A node.

Outputs

If the given node has been nilled (meaning it has the attribute `xsi:nil="true"`), this function returns `true`; otherwise it returns `false`. Calling `nilled( )` with the empty sequence returns the empty sequence.

The `nilled( )` function works only with schema-aware processors, and only if the document is using a schema. The schema can be imported into the stylesheet with `<xsl:import-schema>`, or the XML document can use the `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation` attributes. If no schema is associated with the XML document, all of the nodes in the document will have datatypes, and the `nilled( )` function will always return `false`.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 2, "Accessors."

Example

Here is a very short XML document that we'll use to demonstrate the `nilled( )` function:

```
<?xml version="1.0" encoding="utf-8"?>
 <!-- person.xml -->
 <person
   xmlns="http://www.oreilly.com/xslt"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.oreilly.com/xslt person.xsd">

   <name>Doug Tidwell</name>
   <age>42</age>
   <birthday xsi:nil="true"/>
 </person>
```

Notice that we've defined a schema, *person.xsd*. That schema says that the `<birthday>` element can be nilled:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- person.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="age"/>
        <xs:element ref="birthday"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="name" type="xs:string"/>
  <xs:element name="age" type="xs:positiveInteger"/>
  <xs:element name="birthday" type="xs:date" nillable="true"/>

</xs:schema>
```

The XML document has specified that the `<birthday>` element is nil, and the schema specified that the element is nillable. We'll use this stylesheet to illustrate the `nilled( )` function at work:

```
<?xml version="1.0"?>
<!-- nilled.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:po="http://www.oreilly.com/xslt">

  <xsl:output method="text"/>
  <xsl:import-schema namespace="http://www.oreilly.com/xslt"
    schema-location="person.xsd"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the nilled() function:&#xA;</xsl:text>

    <xsl:for-each select="//*">
      <xsl:text>&#xA;    Element &lt;</xsl:text>
```

```
        <xsl:value-of select="name()"/>
        <xsl:text>&gt; </xsl:text>
        <xsl:value-of select="if (nilled(.))
                              then '===>>> IS nilled!'
                              else 'is not nilled!'"/>
      </xsl:for-each>
    </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
A test of the nilled() function:

    Element <person> is not nilled!
    Element <name> is not nilled!
    Element <age> is not nilled!
    Element <birthday> ===>>> IS nilled!
```

[2.0] node-name()

Returns an expanded QName for nodes that can have names.

Syntax

```
xs:QName? node-name(node()?)
```

Input

A node.

Outputs

For nodes that are allowed to have names, returns the node name. For other kinds of nodes (comment nodes, for example), `node-name( )` returns the empty sequence. If the argument to `node-name( )` is the empty sequence, the function returns the empty sequence, as you'd expect.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 2, "Accessors."

Example

We'll use this stylesheet to illustrate how the `node-name( )` function works with different kinds of nodes:

```
<?xml version="1.0"?>
<!-- node-name.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
```

```
  <xsl:text>&#xA;Here's a test of the node-name() </xsl:text>
  <xsl:text>function:&#xA;</xsl:text>

  <xsl:apply-templates
    select="*|comment()|processing-instruction()|text()"/>
</xsl:template>

<xsl:template match="*">
  <xsl:text>&#xA;</xsl:text>
  <xsl:text>  Element node: node-name(.) = '</xsl:text>
  <xsl:value-of select="node-name(.)"/>
  <xsl:if test="prefix-from-QName(node-name(.))">
    <xsl:text>'&#xA;    Prefix: '</xsl:text>
    <xsl:value-of select="prefix-from-QName(node-name(.))"/>
  </xsl:if>
  <xsl:text>'&#xA;    Local name: '</xsl:text>
  <xsl:value-of select="local-name-from-QName(node-name(.))"/>
  <xsl:if test="namespace-uri-from-QName(node-name(.))">
    <xsl:text>'&#xA;    Namespace URI: '</xsl:text>
    <xsl:value-of select="namespace-uri-from-QName(node-name(.))"/>
  </xsl:if>
  <xsl:text>'&#xA;</xsl:text>

  <xsl:if test="count(@*)">
    <xsl:for-each select="@*">
      <xsl:text>    Attribute node: node-name(.) = '</xsl:text>
      <xsl:value-of select="node-name(.)"/>
      <xsl:text>'&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:if>

  <xsl:for-each select="namespace::*">
    <xsl:call-template name="namespace-node"/>
  </xsl:for-each>

  <xsl:for-each
    select="*|comment()|processing-instruction()|text()">
    <xsl:apply-templates select="."/>
  </xsl:for-each>
</xsl:template>

<xsl:template match="comment()">
  <xsl:text>&#xA;</xsl:text>
  <xsl:text>  Comment node: node-name(.) = '</xsl:text>
  <xsl:value-of select="node-name(.)"/>
  <xsl:text>'&#xA;</xsl:text>
</xsl:template>

<xsl:template match="processing-instruction()">
  <xsl:text>&#xA;</xsl:text>
  <xsl:text>  Processing instruction node: </xsl:text>
  <xsl:text>node-name(.) = '</xsl:text>
  <xsl:value-of select="node-name(.)"/>
  <xsl:text>'&#xA;</xsl:text>
</xsl:template>
```

```
  <xsl:template match="text()">
    <xsl:if test="string-length(normalize-space(.))">
      <xsl:text>  Text node: node-name(.) = '</xsl:text>
      <xsl:value-of select="node-name(.)"/>
      <xsl:text>'&#xA;</xsl:text>
    </xsl:if>
  </xsl:template>

  <xsl:template name="namespace-node">
    <xsl:if test="string(node-name(.)) != 'xml'">
      <xsl:text>  Namespace node: node-name(.) = '</xsl:text>
      <xsl:value-of select="node-name(.)"/>
      <xsl:text>'&#xA;</xsl:text>
    </xsl:if>
  </xsl:template>

</xsl:stylesheet>
```

We'll test this stylesheet with a slightly modified version of our Shakespearean sonnet (we added a processing instruction at the top):

```
<?xml version="1.0"?>
<!-- sonnet-pi.xml -->
<?xml-stylesheet href="node-name.xsl" type="text/xsl"?>
<!-- This is a slightly modified version of our usual sonnet. -->
<sonnet type='Shakespearean'>
  <auth:author xmlns:auth="http://www.authors.com/">
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
    <year-of-death>1616</year-of-death>
  </auth:author>
  <!-- Is there an official title for this sonnet?  They're
       sometimes named after the first line.                  -->
  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
    <line>If hairs be wires, black wires grow on her head.</line>
    <line>I have seen roses damasked, red and white,</line>
    <line>But no such roses see I in her cheeks.</line>
    <line>And in some perfumes is there more delight</line>
    <line>Than in the breath that from my mistress reeks.</line>
    <line>I love to hear her speak, yet well I know</line>
    <line>That music hath a far more pleasing sound.</line>
    <line>I grant I never saw a goddess go,</line>
    <line>My mistress when she walks, treads on the ground.</line>
    <line>And yet, by Heaven, I think my love as rare</line>
    <line>As any she belied with false compare.</line>
  </lines>
</sonnet>
```

Our example is a generic stylesheet that works with any XML document. For our sonnet, it generates this output:

```
Here's a test of the node-name() function:

  Comment node: node-name(.) = ''

  Processing instruction node: node-name(.) = 'xml-stylesheet'

  Comment node: node-name(.) = ''

  Element node: node-name(.) = 'sonnet'
    Local name: 'sonnet'
    Attribute node: node-name(.) = 'type'

  Element node: node-name(.) = 'auth:author'
    Prefix: 'auth'
    Local name: 'author'
    Namespace URI: 'http://www.authors.com/'
  Namespace node: node-name(.) = 'auth'

  Element node: node-name(.) = 'last-name'
    Local name: 'last-name'
  Namespace node: node-name(.) = 'auth'
  Text node: node-name(.) = ''

  Element node: node-name(.) = 'first-name'
    Local name: 'first-name'
  Namespace node: node-name(.) = 'auth'
  Text node: node-name(.) = ''

  ...

  Comment node: node-name(.) = ''

  Element node: node-name(.) = 'title'
    Local name: 'title'
  Text node: node-name(.) = ''

  Element node: node-name(.) = 'lines'
    Local name: 'lines'

  Element node: node-name(.) = 'line'
    Local name: 'line'
  Text node: node-name(.) = ''

  Element node: node-name(.) = 'line'
    Local name: 'line'
  Text node: node-name(.) = ''
  ...
```

Notice that element, attribute, processing instruction, and namespace nodes have names, whereas comment and text nodes don't. Also notice that the document has a processing instruction and a comment node that are outside the root element; those are available in the `match="/"` template.

The node-name( ) function returns an xs:QName. An xs:QName has three parts: the prefix, the local name, and the namespace URI. All elements have a local name, but as you can see from our results here, many elements do not have a prefix or namespace URI. XPath 2.0 provides several functions to work with xs:QNames. In this stylesheet, we use prefix-from-QName( ), local-name-from-QName( ), and namespace-uri-from-QName( ) to get the three parts.

Finally, the <?xml-stylesheet ... processing instruction defines the default stylesheet for processing this XML document. When you invoke an XSLT processor, it typically requires the name of an XML file and a stylesheet. The processor normally uses the stylesheet you specify and ignores the processing instruction. If you want the XSLT processor to use the default stylesheet, there is usually an option to do so. (With Saxon, the -a option does the trick.) Check your processor's documentation for details.

normalize-space()

Removes extra whitespace from its argument string.

Syntax

[1.0] string **normalize-space(***string?***)**
[2.0] xs:string **normalize-space(***xs:string?***)**

Inputs

An optional string. If the argument is omitted, the normalize-space( ) function uses the string value of the context node.

[2.0] If the argument is the empty sequence, normalize-space( ) returns a zero-length string.

Output

The argument string, with whitespace removed as follows:

- All leading whitespace is removed.
- All trailing whitespace is removed.
- Within the string, any sequence of whitespace characters is replaced with a single space.

If the string is all whitespace, a zero-length string is returned.

[2.0] In XSLT 2.0, passing the empty sequence to normalize-space( ) returns the empty sequence.

Defined in

[1.0] XPath section 4.2, "String Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 7.4, "Functions on String Values."

Example

Here is a short example that demonstrates how `normalize-space( )` works:

```
<?xml version="1.0"?>
<!-- normalize-space1.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:variable name="newline">
    <xsl:text>&#xA;</xsl:text>
  </xsl:variable>

  <xsl:variable name="testString">
    <xsl:text>                 This
is


a string
that had        &#x9;&#x9;    lots of

&#xA;&#xA;&#xA;
whitespace.

</xsl:text>
    </xsl:variable>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the normalize-space() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  normalize-space('       </xsl:text>
    <xsl:text>Hello,            World!')="</xsl:text>
    <xsl:value-of
      select="normalize-space('       Hello,            World!')"/>
    <xsl:text>"</xsl:text>
    <xsl:text>&#xA;  normalize-space($newline)="</xsl:text>
    <xsl:value-of select="normalize-space($newline)"/>
    <xsl:text>"&#xA;</xsl:text>
    <xsl:text>  normalize-space($testString)=&#xA;    "</xsl:text>
    <xsl:value-of select="normalize-space($testString)"/>
    <xsl:text>"</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet generates this output:

```
Tests of the normalize-space() function:

  normalize-space('       Hello,            World!')="Hello, World!"
  normalize-space($newline)=""
  normalize-space($testString)=
    "This is a string that had lots of whitespace."
```

In the first and third tests of `normalize-space( )`, the whitespace was removed as advertised, with the intermingled spaces, tabs (`	`) and newlines (`
`) inside the string replaced with a single space. In the second test, the variable has a value of `
`. This is a string containing only whitespace, so the space-normalized version of the string is a zero-length string.

Here's another stylesheet that tests the XSLT 2.0 behavior when calling `normalize-space( )` with the empty sequence as input:

```
<?xml version="1.0"?>
<!-- normalize-space2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the normalize-space() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  normalize-space(())="</xsl:text>
    <xsl:value-of select="normalize-space(())"/>
    <xsl:text>"</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

The results are as exciting as you'd expect:

```
A test of the normalize-space() function:

  normalize-space(())=""
```

[2.0] normalize-unicode()

Given a string, this function returns that string with its characters converted to a particular Unicode normalization form.

Syntax

```
xs:string normalize-unicode(xs:string?)
xs:string normalize-unicode(xs:string?, $normalizationForm as xs:string)
```

Inputs

The string to be normalized. You can optionally specify a second string naming a normalization form. The Unicode normalization forms are `NFC`, `NFD`, `NFKC`, `NFKD`, and `FULLY-NORMALIZED`.

In general, the Unicode normalization forms deal with how combining characters are processed. As an example, the A-ring character (Å) can be a single character (`Å`) or an uppercase A (`A`) followed by a combining ring above character (`̊`). These two representations are semantically equivalent, but different at a character level. Unicode normalization converts the different representations to the same form so they can be compared.

All conforming processors must support the NFC format. They are also free to support any or all of the other formats defined by Unicode, and they can support their own formats if they want.

If you'd like to know more, see Unicode Standard Annex #15, Unicode Normalization Forms, available at *http://www.unicode.org/unicode/reports/tr15/*.

Outputs

The input string, with its characters converted to the appropriate Unicode normalization form. If the value of the first argument is the empty sequence, normalize-unicode( ) returns a zero-length string.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 7.4, "Functions on String Values."

Example

Here is a very short example that compares the two character strings just mentioned:

```
<?xml version="1.0"?>
<!-- normalize-unicode.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Here is a test of the normalize-</xsl:text>
    <xsl:text>unicode() function:&#xA;</xsl:text>

    <xsl:text>&#xA;  compare('&#xC5', </xsl:text>
    <xsl:text>'&#x41;&#x30A;') = </xsl:text>
    <xsl:value-of select="compare('&#xC5;', '&#x41;&#x30A;')"/>

    <xsl:text>&#xA;&#xA;  compare(normalize-unicode</xsl:text>
    <xsl:text>('&#xC5'),&#xA;          normalize-unicode(</xsl:text>
    <xsl:text>'&#x41;&#x30A;')) = </xsl:text>
    <xsl:value-of
      select="compare(normalize-unicode('&#xC5;'),
              normalize-unicode('&#x41;&#x30A;'))"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Here is a test of the normalize-unicode() function:

  compare('&#xC5', '&#x41;&#x30A;') = 1

  compare(normalize-unicode('&#xC5'),
          normalize-unicode('&#x41;&#x30A;')) = 0
```

In the first comparison, the two strings are not equal. In the second comparison, the Unicode normalized versions of the string are equal.

not()

Returns the negation of its argument. If the argument is not a boolean value already, it is converted to a boolean value using the rules described in the `boolean( )` function entry.

Syntax

```
[1.0] boolean not(boolean)
[2.0] xs:boolean not(item()*)
```

Inputs

A boolean value, or more commonly, an XPath expression that evaluates to a boolean value.

Output

`false` if the input parameter is `true`; `true` if the input parameter is `false`.

Defined in

[1.0] XPath section 4.3, "Boolean Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 9.3, "Functions on Boolean Values."

Example

To demonstrate the `not( )` function, we'll use the same stylesheet and XML document we used for the `boolean( )` function. Here's our XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
```

```
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

We'll process this document with the following stylesheet, which uses `not( )` instead of `boolean( )`. The `boolean( )` function converts its argument to a boolean value; the `not( )` function implicitly uses `boolean( )` to convert its argument to a boolean value, and then negates it.

```
<?xml version="1.0"?>
<!-- not.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the not() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  not((true()) = </xsl:text>
    <xsl:value-of select="not(true())"/>

    <xsl:text>&#xA;  not(true) = </xsl:text>
    <xsl:value-of select="not(true)"/>

    <xsl:text>&#xA;  not('false') = </xsl:text>
    <xsl:value-of select="not('false')"/>

    <xsl:text>&#xA;  not('7') = </xsl:text>
    <xsl:value-of select="not('7')"/>

    <xsl:text>&#xA;  not(/report/brand/units[. > 20000]) = </xsl:text>
    <xsl:value-of select="not(/report/brand/units[. > 20000])"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Tests of the not() function:

  not((true()) = false
  not(true) = true
  not('false') = false
  not('7') = false
  not(/report/brand/units[. > 20000]) = false
```

As you'd expect, these results are the exact opposite of the results we got when we tested the `boolean( )` function. For the first test, the argument is a call to the `true( )` function, which always returns `true`. The negation of `true` is `false`. For the second test, we're asking for all the `<true>` elements in the current context; there are no `<true>` elements, so this is `false` and its negation is `true`. The argument of the third test is a string whose length is greater than zero, so `not( )` returns `false` (a nonzero-length string is `true`). Similarly, the argument of the fourth test is a string whose length is greater than zero, so `not( )` returns `false` here as well.

The final example uses an XPath statement to select all of the <units> elements in our sales report that have a numeric value greater than 20000. Because this selects at least one node, the expression is true and not( ) returns false. This works in XSLT 1.0 because it generates a node-set with at least one member, and it works in XSLT 2.0 because it creates a sequence whose first item is a node. If we change the expression to look for sales figures greater than 30000, not( ) would return true.

number()

Converts its argument to a number.

Syntax

```
[1.0] number number(object?)
[2.0] xs:double number()
[2.0] xs:double number(xs:anyAtomicType)
```

Inputs

[1.0] An object.

[2.0] An xs:anyAtomicType value.

Output

A number. If no argument is passed to the number function, the context item is used. The specific rules for converting the argument to a number are different for XSLT 1.0 and 2.0.

[1.0] Here are the rules for XSLT 1.0:

- If the argument is a boolean value, the value true is converted to the number 1; the value false is converted to the number 0.
- If the argument is a node-set, the node-set is converted to a string as if it were passed to the string( ) function, and then that string is converted to a number like any other string. (Remember that the string( ) function returns the string value of the first node in the node-set.)
- If the argument is a string, it is converted as follows:
  - —If the string consists of optional whitespace, followed by an optional minus sign (-), followed by a number, followed by whitespace, it is converted to the floating-point value nearest to the mathematical value represented by the string. (The IEEE 754 standard defines a round-to-nearest rule; see the standard for more information.)
  - —Any other string is converted to the value NaN (not a number).
- XSLT 1.0 processors are allowed to support other datatypes. If the argument is any other type, it is converted to a number in a way that depends on that type. See the documentation for your XSLT processor to find out what other types are supported and how they are converted to numbers.

[2.0] Because XSLT 2.0 supports more datatypes, the rules are more complicated:

- If the argument is already an `xs:double`, it is returned unchanged.
- If the argument is an `xs:boolean`, `true` is returned as `1.0E0` and `false` is returned as `0.0E0`.
- If the argument an `xs:float`, `xs:decimal`, or `xs:integer`, it is converted to an `xs:double` and returned. If the argument is an `xs:float` and is one of the `xs:float` values `INF`, `-INF`, `NaN`, positive zero, or negative zero, it is returned as the `xs:double` version of those values.
- If the argument is the empty sequence, `NaN` (not a number) is returned.
- If the argument (or the context node, if there isn't an argument) can't be converted to an `xs:double`, `NaN` is returned. (The `xs:date`, `xs:dateTime`, and `xs:time` datatypes can't be converted to numbers, for example.)

Defined in

[1.0] XPath section 4.4, "Number Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 14, "Functions and Operators on Nodes."

Example

We'll use our document of chocolate bar sales to illustrate the `number( )` function:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

We'll test the `number( )` function with a variety of arguments:

```
<?xml version="1.0"?>
<!-- number.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the number() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  number(true()) = </xsl:text>
    <xsl:value-of select="number(true())"/>
    <xsl:text>&#xA;&#xA;  number(false()) = </xsl:text>
    <xsl:value-of select="number(false())"/>
    <xsl:text>&#xA;&#xA;  number(/report/brand[2]/units) = </xsl:text>
    <xsl:value-of select="number(/report/brand[2]/units)"/>
    <xsl:text>&#xA;&#xA;  number(//units) = </xsl:text>
    <xsl:value-of select="number(//units)"/>
    <xsl:text>&#xA;&#xA;  number(/report/title) = </xsl:text>
    <xsl:value-of select="number(/report/title)"/>
  </xsl:template>

</xsl:stylesheet>
```

The output of our stylesheet looks like this:

```
Tests of the number() function:

  number(true()) = 1

  number(false()) = 0

  number(/report/brand[2]/units) = 8203

  number(//units) = 27408

  number(/report/title) = NaN
```

This is an XSLT 1.0 stylesheet, so the expression `number(//units)` works in an XSLT 2.0 processor running in XSLT 1.0 compatibility mode. In XSLT 2.0, the processor raises an error if we pass more than one node to the `number( )` function. Here's a slightly different stylesheet designed for XSLT 2.0:

```
<?xml version="1.0"?>
<!-- number2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the number() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  number(true()) = </xsl:text>
    <xsl:value-of select="number(true())"/>
    <xsl:text>&#xA;&#xA;  number(false()) = </xsl:text>
```

```
    <xsl:value-of select="number(false())"/>
    <xsl:text>&#xA;&#xA;  number(/report/brand[2]/units) = </xsl:text>
    <xsl:value-of select="number(/report/brand[2]/units)"/>
    <xsl:text>&#xA;&#xA;  number((//units)[1]) = </xsl:text>
    <xsl:value-of select="number((//units)[1])"/>
    <xsl:text>&#xA;&#xA;  number(/report/title) = </xsl:text>
    <xsl:value-of select="number(/report/title)"/>
    <xsl:text>&#xA;&#xA;  number(current-date()) = </xsl:text>
    <xsl:value-of select="number(current-date())"/>
    <xsl:text>&#xA;&#xA;  number(current-time()) = </xsl:text>
    <xsl:value-of select="number(current-time())"/>
  </xsl:template>

</xsl:stylesheet>
```

Notice that we had to change the `number(//units)` call to `number((//units)[1])`. This retrieves all of the `<units>` elements at all levels of the document and selects the first one. At any rate, here are the results:

```
Tests of the number() function:

  number(true()) = 1

  number(false()) = 0

  number(/report/brand[2]/units) = 8203

  number((//units)[1]) = 27408

  number(/report/title) = NaN

  number(current-date()) = NaN

  number(current-time()) = NaN
```

Notice that the `xs:date` and `xs:time` values are returned as `NaN`.

[2.0] one-or-more()

Raises an error unless its argument is a sequence containing one or more items. This function can be used as a form of type checking. For example, we might have an `<addressbook>` element that's required to have at least one `<address>` element. We could call `one-or-more(address)` with an `<addressbook>` element as the current item; if at least one `<address>` element isn't present, the XSLT processor would raise an error.

Syntax

```
item()+ one-or-more(item()*)
```

Inputs

A sequence. If that sequence doesn't have at least one item, `one-or-more( )` raises an error.

Outputs

Assuming the input sequence has at least one item, `one-or-more( )` returns that sequence. If the input sequence is the empty sequence, `one-or-more( )` raises an error.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.2, "Functions That Test the Cardinality of Sequences." More details about this function can be found in XQuery 1.0 and XPath 2.0 Formal Semantics section 7.2, "Standard Functions with Specific Static Typing Rules."

Example

Here's a simple stylesheet that illustrates the `one-or-more( )` function:

```
<?xml version="1.0"?>
<!-- one-or-more.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:variable name="testSequence" as="item()*">
      <xsl:sequence
        select="(3, 4, 5, current-date(), current-time(), 8, 'blue',
                'red', xs:float(3.14), 42, xs:date('1995-04-21'))"/>
    </xsl:variable>

    <xsl:text>&#xA;Here is a test of the one-or-more() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>&#xA;  Our sequence is:&#xA;&#xA;    </xsl:text>
    <xsl:value-of select="$testSequence" separator="&#xA;    "/>

    <xsl:if test="count(one-or-more($testSequence))">
      <xsl:text>&#xA;&#xA;  Our test sequence has one or </xsl:text>
      <xsl:text>more items!</xsl:text>
    </xsl:if>

  </xsl:template>

</xsl:stylesheet>
```

Here are the results of our stylesheet:

```
Here is a test of the one-or-more() function:

  Our sequence is:

    3
    4
    5
```

```
2006-07-22-04:00
23:04:38.183-04:00
8
blue
red
3.14
42
1995-04-21

Our test sequence has one or more items!
```

Notice that we use the `count( )` function in the `test` attribute of the `<xsl:if>` element. At first glance, it seems like this would be a reasonable way to write the `<xsl:if>` element:

```
<xsl:if test="one-or-more($testSequence)">  <!-- doesn't work! -->
```

The `one-or-more( )` function returns the input sequence, assuming it has at least one value. The XSLT processor attempts to convert the contents of the `test` attribute to a boolean. If the sequence returned by `one-or-more( )` is a single item *or* its first item is a node, this works. However, our example sequence would cause the XSLT processor to raise an error. A sequence of more than one item whose first item is an atom can't be converted to a boolean value. That's why we have to write the `<xsl:if>` element like this:

```
<xsl:if test="count(one-or-more($testSequence))"/>
```

The XSLT processor converts the numeric value returned by `count( )` into a boolean value. That value will always be at least `1`, so the `test` attribute evaluates to `true` anytime the `one-or-more( )` function works.

See Also

The descriptions of the [2.0] `exactly-one( )` and [2.0] `zero-or-one( )` functions.

position()

Returns the position of the context item within the sequence of items currently being processed.

Syntax

```
[1.0] number position()
[2.0] xs:integer position()
```

Inputs

None.

Output

A number equal to the position of the current node in the sequence of items currently being processed. For example, if the current node is the fifth `<li>` being processed, `position( )` returns `5`.

Defined in

[1.0] XPath section 4.1, "Node Set Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 16, "Context Functions."

Examples

This example uses the `position( )` function to create a `style` attribute for the cells of an HTML table. The background colors cycle through the options `black`, `gray`, and `white`, while the foreground colors cycle through `white`, `white`, and `black`. Here's the XML document we'll use:

```
<?xml version="1.0"?>
<!-- albums.xml -->
<list xml:lang="en">
  <title>Albums I've bought recently:</title>
  <listitem>The Sacred Art of Dub</listitem>
  <listitem>Only the Poor Man Feel It</listitem>
  <listitem>Excitable Boy</listitem>
  <listitem xml:lang="sw">Aki Special</listitem>
  <listitem xml:lang="en-gb">Combat Rock</listitem>
  <listitem xml:lang="zu">Talking Timbuktu</listitem>
  <listitem xml:lang="jz">The Birth of the Cool</listitem>
</list>
```

We'll use this stylesheet to generate our HTML document:

```
<?xml version="1.0"?>
<!-- position.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>
          <xsl:value-of select="/list/title"/>
        </title>
      </head>
      <body style="font-family: sans-serif;">
        <h1>
          <xsl:value-of select="/list/title"/>
        </h1>
        <table border="1" cellpadding="5">
          <xsl:for-each select="/list/listitem">
            <xsl:variable name="style">
              <xsl:text>color: </xsl:text>
              <xsl:choose>
                <xsl:when test="position() mod 3 = 1">white</xsl:when>
                <xsl:when test="position() mod 3 = 2">white</xsl:when>
                <xsl:otherwise>black</xsl:otherwise>
              </xsl:choose>
              <xsl:text>; background: </xsl:text>
              <xsl:choose>
                <xsl:when test="position() mod 3 = 1">black</xsl:when>
```

```
            <xsl:when test="position() mod 3 = 2">gray</xsl:when>
            <xsl:otherwise>white</xsl:otherwise>
          </xsl:choose>
          <xsl:text>;</xsl:text>
        </xsl:variable>
        <tr style="{$style}">
          <td>
            <b><xsl:value-of select="."/></b>
          </td>
        </tr>
      </xsl:for-each>
    </table>
  </body>
 </html>
</xsl:template>

</xsl:stylesheet>
```

We use the `position( )` function to cycle through the different background and foreground colors. Our stylesheet generates the following results:

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Albums I've bought recently:</title>
   </head>
   <body style="font-family: sans-serif;">
      <h1>Albums I've bought recently:</h1>
      <table border="1" cellpadding="5">
         <tr style="color: white; background: black;">
            <td><b>The Sacred Art of Dub</b></td>
         </tr>
         <tr style="color: white; background: gray;">
            <td><b>Only the Poor Man Feel It</b></td>
         </tr>
         <tr style="color: black; background: white;">
            <td><b>Excitable Boy</b></td>
         </tr>
         <tr style="color: white; background: black;">
            <td><b>Aki Special</b></td>
         </tr>
         <tr style="color: white; background: gray;">
            <td><b>Combat Rock</b></td>
         </tr>
         <tr style="color: black; background: white;">
            <td><b>Talking Timbuktu</b></td>
         </tr>
         <tr style="color: white; background: black;">
            <td><b>The Birth of the Cool</b></td>
         </tr>
      </table>
   </body>
</html>
```

When rendered, the HTML file looks like Figure C-5.

Figure C-5. HTML file displaying items with different background colors

[2.0] prefix-from-QName()

Given an `xs:QName` value, returns its namespace prefix.

Syntax

```
xs:NCName? prefix-from-QName(xs:QName?)
```

Input

An `xs:QName` value.

Output

An `xs:NCName` ("noncolonized" name) representing the prefix from the `QName`. If the `QName` doesn't have a prefix or the `xs:QName` argument is the empty sequence, `prefix-from-QName( )` returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 11.2, "Operators and Functions Related to QNames."

Example

As usual, when we're working with namespaces, we'll use our Shakespearean sonnet as our XML input document:

```
<?xml version="1.0"?>
<!-- sonnet.xml -->
```

```
<sonnet type='Shakespearean'>
  <auth:author xmlns:auth="http://www.authors.com/">
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
    <year-of-death>1616</year-of-death>
  </auth:author>
  <!-- Is there an official title for this sonnet?  They're
       sometimes named after the first line.                    -->
  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
    <line>If hairs be wires, black wires grow on her head.</line>
    <line>I have seen roses damasked, red and white,</line>
    <line>But no such roses see I in her cheeks.</line>
    <line>And in some perfumes is there more delight</line>
    <line>Than in the breath that from my mistress reeks.</line>
    <line>I love to hear her speak, yet well I know</line>
    <line>That music hath a far more pleasing sound.</line>
    <line>I grant I never saw a goddess go,</line>
    <line>My mistress when she walks, treads on the ground.</line>
    <line>And yet, by Heaven, I think my love as rare</line>
    <line>As any she belied with false compare.</line>
  </lines>
</sonnet>
```

Here is our stylesheet:

```
<?xml version="1.0"?>
<!-- prefix-from-qname.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the prefix-from-QName() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:variable name="testQName1" as="xs:QName"
      select="resolve-QName('auth:last-name',
              /sonnet/*:author/last-name)"/>
    <xsl:text>&#xA;  testQName1 = resolve-QName('auth:last-name', </xsl:text>
    <xsl:text>/sonnet/*:author/last-name)</xsl:text>
    <xsl:text>&#xA;  prefix-from-QName($testQName1) = "</xsl:text>
    <xsl:value-of select="prefix-from-QName($testQName1)"/>
    <xsl:text>"</xsl:text>

    <xsl:variable name="testQName2" as="xs:QName"
      select="resolve-QName('something-else', /sonnet)"/>
    <xsl:text>&#xA;&#xA;  testQName2 = resolve-QName('</xsl:text>
    <xsl:text>something-else', /sonnet)</xsl:text>
```

```
    <xsl:text>&#xA;  prefix-from-QName($testQName2) = "</xsl:text>
    <xsl:value-of select="prefix-from-QName($testQName2)"/>
    <xsl:text>"</xsl:text>

    <xsl:variable name="testQName3" as="xs:QName"
      select="QName('http://www.authors.com/', 'pfx:writer')"/>
    <xsl:text>&#xA;&#xA;  testQName3 = QName('http://www.</xsl:text>
    <xsl:text>authors.com/', 'pfx:writer')&#xA;</xsl:text>
    <xsl:text>  prefix-from-QName($testQName3) = "</xsl:text>
    <xsl:value-of select="prefix-from-QName($testQName3)"/>
    <xsl:text>"</xsl:text>

  </xsl:template>

</xsl:stylesheet>
```

And here are our results:

```
Tests of the prefix-from-QName() function:

  testQName1 = resolve-QName('auth:last-name', /sonnet/*:author/last-name)
  prefix-from-QName($testQName1) = "auth"

  testQName2 = resolve-QName('something-else', /sonnet)
  prefix-from-QName($testQName2) = ""

  testQName3 = QName('http://www.authors.com/', 'pfx:writer')
  prefix-from-QName($testQName3) = "pfx"
```

The argument to the `prefix-from-QName( )` function *must* be an `xs:QName`. In this stylesheet, we create `QName`s with the `resolve-QName( )` function and the `QName( )` constructor function.

For the first test, we create a `QName` with a prefixed element name of `auth:last-name`. The context for resolving the `QName` (the second argument to `resolve-QName( )`) is the `<last-name>` element in our sonnet. The `resolve-QName( )` function returns a `QName` with a local name of `last-name`, a namespace prefix of `auth`, and a namespace URI of `http://www.authors.com/`. If there are no nodes that match the XPath expression `/sonnet/*:author/last-name`, the call to `resolve-QName( )` fails. The call also fails if the `auth` prefix is not defined for the `<last-name>` element.

The second test of `prefix-from-QName( )` uses an unprefixed element name in the call to `resolve-QName( )`. The XML document doesn't have a default namespace, so our `QName` has an empty string for its namespace prefix and namespace URI.

For the final test, we use the `QName( )` constructor function to create a `QName` from scratch. The two arguments to the constructor function are the namespace URI and an element name—in this case, a prefixed element name. The resulting `QName` has a local name of `writer`, a namespace prefix of `pfx`, and a namespace URI of `http://www.authors.com/`.

[2.0] QName()

This is a constructor function that allows you to create a new `QName`.

Syntax

```
xs:QName QName($paramURI as xs:string?, $paramQName as xs:string)
```

Inputs

Two `xs:string`s. The first string is the namespace URI of the `QName`, and the second is the string value of the `QName`. If the second string contains a colon (`:`), the substring before the colon is the namespace prefix of the `QName` and the substring after the colon is the local name. For example, if the second string is `auth:author`, the namespace prefix is `auth` and the local name is `author`.

The namespace URI string can be a zero-length string or the empty sequence, meaning this `QName` does not belong to a namespace. In this case, the XSLT processor raises an error if the second string contains a colon. (In other words, you can't have a namespace prefix if you don't have a namespace URI.)

Output

An `xs:QName` value.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 11.1, "Additional Constructor Functions for QNames."

Example

Here is a stylesheet that creates several `QName`s, and then uses various functions to print the details of those `QName`s:

```
<?xml version="1.0"?>
<!-- qname.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the QName() constructor function:&#xA;</xsl:text>

    <xsl:variable name="testQName1" as="xs:QName"
      select="QName('http://www.authors.com', 'auth:author')"/>

    <xsl:text>&#xA;  testQName1 = QName('http://www.</xsl:text>
    <xsl:text>authors.com', 'auth:author')</xsl:text>
    <xsl:text>&#xA;  prefix-from-QName</xsl:text>
    <xsl:text>($testQName1) = "</xsl:text>
    <xsl:value-of select="prefix-from-QName($testQName1)"/>
    <xsl:text>"&#xA;  local-name-from-QName($testQName1) = "</xsl:text>
    <xsl:value-of select="local-name-from-QName($testQName1)"/>
    <xsl:text>"&#xA;  namespace-uri-from-QName($testQName1) = "</xsl:text>
    <xsl:value-of select="namespace-uri-from-QName($testQName1)"/>
    <xsl:text>"</xsl:text>
```

```
    <xsl:variable name="testQName2" as="xs:QName"
      select="QName('', 'sonnet')"/>

    <xsl:text>&#xA;&#xA;  testQName2 = QName('', 'sonnet')</xsl:text>
    <xsl:text>&#xA;  prefix-from-QName</xsl:text>
    <xsl:text>($testQName2) = "</xsl:text>
    <xsl:value-of select="prefix-from-QName($testQName2)"/>
    <xsl:text>"&#xA;  local-name-from-QName($testQName2) = "</xsl:text>
    <xsl:value-of select="local-name-from-QName($testQName2)"/>
    <xsl:text>"&#xA;  namespace-uri-from-QName($testQName2) = "</xsl:text>
    <xsl:value-of select="namespace-uri-from-QName($testQName2)"/>
    <xsl:text>"</xsl:text>

    <xsl:variable name="testQName3" as="xs:QName"
      select="QName('http://www.authors.com/', 'writer')"/>

    <xsl:text>&#xA;&#xA;  testQName3 = QName('http://www.</xsl:text>
    <xsl:text>authors.com/', 'writer')</xsl:text>
    <xsl:text>&#xA;  prefix-from-QName</xsl:text>
    <xsl:text>($testQName3) = "</xsl:text>
    <xsl:value-of select="prefix-from-QName($testQName3)"/>
    <xsl:text>"&#xA;  local-name-from-QName($testQName3) = "</xsl:text>
    <xsl:value-of select="local-name-from-QName($testQName3)"/>
    <xsl:text>"&#xA;  namespace-uri-from-QName($testQName3) = "</xsl:text>
    <xsl:value-of select="namespace-uri-from-QName($testQName3)"/>
    <xsl:text>"</xsl:text>

  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Tests of the QName() constructor function:

  testQName1 = QName('http://www.authors.com', 'auth:author')
  prefix-from-QName($testQName1) = "auth"
  local-name-from-QName($testQName1) = "author"
  namespace-uri-from-QName($testQName1) = "http://www.authors.com"

  testQName2 = QName('', 'sonnet')
  prefix-from-QName($testQName2) = ""
  local-name-from-QName($testQName2) = "sonnet"
  namespace-uri-from-QName($testQName2) = ""

  testQName3 = QName('http://www.authors.com/', 'writer')
  prefix-from-QName($testQName3) = ""
  local-name-from-QName($testQName3) = "writer"
  namespace-uri-from-QName($testQName3) = "http://www.authors.com/"
```

Our stylesheet creates three `QName`s, and then retrieves the namespace prefix, local name, and namespace URI for each one. For the first test, we create a `QName` with a prefixed element name of `auth:last-name`. The second test has an empty namespace and an unprefixed element name, and the third test has a namespace URI but no element prefix.

[2.0] regex-group()

When using a regular expression to process a string, this function returns portions of the analyzed string based on groups in the regular expression. This function is used inside the `<xsl:matching-substring>` element only.

Syntax

```
xs:string regex-group(xs:integer)
```

Inputs

An `xs:integer` representing a section of the regular expression.

Output

The portion of the analyzed string that matches the specified portion of the regular expression. A portion of a regular expression must be in parentheses to be considered a group.

Here are some notes about how `regex-group( )` works:

- Calling `regex-group(0)` returns the entire matching substring, including characters that don't belong to any group.
- If the requested portion of the regular expression exists, but doesn't match anything in the analyzed string, `regex-group( )` returns a zero-length string. (In other words, the group is optional in the regular expression.)
- If the requested portion of the regular expression exists, but it matches the zero-length portion of the analyzed string, `regex-group( )` returns a zero-length string.
- If the requested portion of the regular expression doesn't exist (we call `regex-group(4)` when the regular expression has only three groups, for example), `regex-group( )` returns a zero-length string.
- Finally, `regex-group( )` returns a zero-length string if the group number is a negative integer.

Defined in

XSLT 2.0 section 15, "Regular Expressions."

Example

We'll use a simplified version of one of the stylesheets we used to demonstrate the `<xsl:analyze-string>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- regex-group.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
```

```
    <xsl:text>Formatted phone numbers:&#xA;</xsl:text>
    <xsl:for-each select="phonelist/phonenumber">
      <xsl:analyze-string select="."
        regex="([0-9]{{3}})-([0-9]{{3}})-([0-9]{{4}})">
        <xsl:matching-substring>
          <xsl:text>&#xA;  The data </xsl:text>
          <xsl:value-of select="regex-group(0)"/>
          <xsl:text> matches.  </xsl:text>
          <xsl:text>&#xA;    The formatted phone number is: +1 (</xsl:text>
          <xsl:value-of select="regex-group(1)"/>
          <xsl:text>) </xsl:text>
          <xsl:value-of select="regex-group(2)"/>
          <xsl:text>-</xsl:text>
          <xsl:value-of select="regex-group(3)"/>
        </xsl:matching-substring>
      </xsl:analyze-string>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

We're using a regular expression to analyze telephone numbers. The regular expression `([0-9]{{3}})-([0-9]{{3}})-([0-9]{{4}})` contains three groups: `([0-9]{{3}})`, `([0-9]{{3}})`, and `([0-9]{{4}})`. The two dashes between the groups are not in parentheses, so they are not part of any group.

We'll use this stylesheet to process this very simple document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- simple-phonelist.xml -->
<phonelist>
  <phonenumber>Doug's number is 919-555-1212</phonenumber>
  <phonenumber>617-555-1212</phonenumber>
</phonelist>
```

And here are the results:

```
Formatted phone numbers:

  The data 919-555-1212 matches.
    The formatted phone number is: +1 (919) 555-1212
  The data 617-555-1212 matches.
    The formatted phone number is: +1 (617) 555-1212
```

For each matching string, `regex-group(0)` returns the entire matching string, `regex-group(1)` returns the area code (we're using the Canadian and U.S. phone number format here), `regex-group(2)` returns the exchange and `regex-group(3)` returns the last four digits of the phone number. Notice that the output of `regex-group(0)` is the entire matching string, including the two dashes that don't belong to any matching group. In the first line of the XML source file, `regex-group(0)` *doesn't* return the characters outside the matching string, just the part of the string that matches the regular expression.

Finally, if our regular expression allows a two- or three-letter state or province abbreviation at the end of the phone number (we used the regular expression `([0-9]{{3}})-([0-9]{{3}})-([0-9]{{4}})( [A-Z]{{2,3}})?`), `regex-group(4)` would return a zero-length string for both

<phonenumber> elements in our sample document. Although the match technically exists, it doesn't contain any data. In this case, we could use string-length(regex-group(4)) to see whether the match contains any data.

See Also

The definition of the [2.0] <xsl:analyze-string> element.

[2.0] remove()

Given a sequence and a position, removes the item at the requested position.

Syntax

```
item()* remove($target as item()*, $position as xs:integer)
```

Inputs

A sequence and an xs:integer.

Outputs

This function returns the input sequence with the item at the requested position removed. If the position is less than 1 or greater than the length of the sequence, the sequence is returned unchanged. Finally, if the sequence is the empty sequence, the empty sequence is returned.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.1, "General Functions and Operators on Sequences."

Example

To illustrate the remove( ) function, we'll look at three short stylesheets. Each stylesheet uses a different type of sequence. The first stylesheet uses a singleton sequence:

```
<?xml version="1.0"?>
<!-- remove1.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:variable name="singleton" as="item()*">
      <xsl:sequence select="(3)"/>
    </xsl:variable>

    <xsl:text>&#xA;Here is a test of the remove() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>&#xA;  Our sequence ($singleton) </xsl:text>
```

```
    <xsl:text>is:&#xA;      </xsl:text>
    <xsl:value-of select="$singleton" separator="&#xA;    "/>
    <xsl:text>&#xA;&#xA;    remove($singleton, 0) = </xsl:text>
    <xsl:value-of select="remove($singleton, 0)"
      separator="&#xA;    "/>
    <xsl:text>&#xA;&#xA;    remove($singleton, 7) = </xsl:text>
    <xsl:value-of select="remove($singleton, 7)"
      eparator="&#xA;    "/>
    <xsl:text>&#xA;&#xA;    remove($singleton, 1) = </xsl:text>
    <xsl:value-of select="remove($singleton, 1)"
      separator="&#xA;    "/>
    <xsl:text>&#xA;&#xA;    empty(remove($singleton, 1)) = </xsl:text>
    <xsl:value-of select="empty(remove($singleton, 1))"/>

  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Here is a test of the remove() function:

  Our sequence ($singleton) is:
      3

    remove($singleton, 0) = 3

    remove($singleton, 7) = 3

    remove($singleton, 1) =

    empty(remove($singleton, 1)) = true
```

In our results, removing an item at positions `0` and `7` doesn't change the sequence at all. Removing the first item from a singleton returns the empty sequence, as you would expect. As a final test, we use the `empty( )` function to show that removing one item from a singleton does, in fact, return the empty sequence.

Our next test of the `remove( )` function uses a longer sequence that contains a variety of datatypes:

```
<?xml version="1.0"?>
<!-- remove2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:variable name="longSequence" as="item()*">
      <xsl:sequence
        select="(3, 4, 5, current-date(), current-time(), 8, 'blue',
                 'red', xs:float(3.14), 42, xs:date('1995-04-21'))"/>
    </xsl:variable>
```

```
    <xsl:text>&#xA;Here's another test of the remove() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>&#xA;  Our sequence ($longSequence) </xsl:text>
    <xsl:text>is:&#xA;      </xsl:text>
    <xsl:value-of select="$longSequence" separator="&#xA;      "/>
    <xsl:text>&#xA;&#xA;    Test 1. remove($longSequence, 1) </xsl:text>
    <xsl:text>= &#xA;      </xsl:text>
    <xsl:value-of select="remove($longSequence, 1)"
      separator="&#xA;      "/>
    <xsl:text>&#xA;&#xA;    Test 2. remove($longSequence, 1) =</xsl:text>
    <xsl:text>&#xA;        [We're running this test again to show the </xsl:text>
    <xsl:text>variable didn't change]&#xA;      </xsl:text>
    <xsl:value-of select="remove($longSequence, 1)"
      separator="&#xA;      "/>
    <xsl:text>&#xA;&#xA;    Test 3. remove(remove($longSequence, 7), 1) </xsl:text>
    <xsl:text>= &#xA;      </xsl:text>
    <xsl:value-of select="remove(remove($longSequence, 7), 1)"
      separator="&#xA;      "/>

    <xsl:text>&#xA;&#xA;    Creating a new variable based on </xsl:text>
    <xsl:text>remove($longSequence, 1): &#xA;      </xsl:text>
    <xsl:text>updatedSequence = remove($longSequence, 1): &#xA;      </xsl:text>
    <xsl:variable name="updatedSequence" as="item()*"
      select="remove($longSequence, 1)"/>
    <xsl:text>&#xA;    Test 4. $updatedSequence = &#xA;      </xsl:text>
    <xsl:value-of select="$updatedSequence" separator="&#xA;      "/>

  </xsl:template>

</xsl:stylesheet>
```

The somewhat longwinded results look like this:

```
Here's another test of the remove() function:

  Our sequence ($longSequence) is:
      3
      4
      5
      2006-07-25-04:00
      03:14:18.783-04:00
      8
      blue
      red
      3.14
      42
      1995-04-21

    Test 1. remove($longSequence, 1) =
      4
      5
      2006-07-25-04:00
      03:14:18.783-04:00
      8
```

```
  blue
  red
  3.14
  42
  1995-04-21

Test 2. remove($longSequence, 1) =
    [We're running this test again to show the variable didn't change]
  4
  5
  2006-07-25-04:00
  03:14:18.783-04:00
  8
  blue
  red
  3.14
  42
  1995-04-21

Test 3. remove(remove($longSequence, 7), 1) =
  4
  5
  2006-07-25-04:00
  03:31:45.859-04:00
  8
  red
  3.14
  42
  1995-04-21

Creating a new variable based on remove($longSequence, 1):
  updatedSequence = remove($longSequence, 1):

Test 4. $updatedSequence =
  4
  5
  2006-07-25-04:00
  03:14:18.783-04:00
  8
  blue
  red
  3.14
  42
  1995-04-21
```

Our stylesheet begins by listing the items in the sequence. In our first test, removing the first item returns a sequence in which the first item (`3`) has been removed. Next, we call `remove( )` against the same sequence. The results are the same. Remember, variables in XSLT don't change; they can be initialized, but not changed. We can call `remove($longSequence, 1)` as many times as we want, but the results will always be the same. If we want to "remove" an item permanently, we need to store the sequence returned by `remove( )` in a variable.

In our third test we nest calls to `remove( )` to remove the first and seventh items. If we wanted to remove the first two items, it would be more efficient to use `subsequence($longSequence,`

3). Because we're removing items that aren't adjacent, the nested calls to remove( ) are the way to go.

The fourth example stores the results of the remove( ) function in a variable. That variable's type is item( )* as well. Here are three ways of creating the new variable:

```
<!-- Simplest approach -->
<xsl:variable name="updatedSequence1" as="item()*"
  select="remove($longSequence, 1)"/>

<!-- This works... -->
<xsl:variable name="updatedSequence2" as="item()*">
  <xsl:sequence select="remove($longSequence, 1)"/>
</xsl:variable>

<!-- Doesn't work! -->
<xsl:variable name="updatedSequence3" as="item()*">
  <xsl:value-of select="remove($longSequence, 1)"/>
</xsl:variable>
```

The first approach is the simplest; it's what we used in our stylesheet. The second approach works and is useful if we need to add multiple items to this sequence. *The third approach doesn't work.* The <xsl:value-of> element inserts *the string value* of the subsequence, with the values separated by spaces. This creates a sequence of one item:

```
$updatedSequence3 =
4 5 2006-07-25-04:00 03:20:35.815-04:00 8 blue red 3.14 42 1995-04-21
```

Using <xsl:value-of> seems like a reasonable approach, but it doesn't work. Be sure to use <xsl:sequence> instead. (I'm mentioning that here because I've made the same mistake; hopefully, you can learn from my absent-mindedness.)

Our last example illustrates remove( ) with the empty sequence:

```
<?xml version="1.0"?>
<!-- remove3.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:variable name="emptySequence" as="item()*">
      <xsl:sequence select="()"/>
    </xsl:variable>

    <xsl:text>&#xA;Here's a final test of the remove() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>&#xA;  Our sequence ($emptySequence) is </xsl:text>
    <xsl:text>empty:&#xA;      empty($emptySequence) = </xsl:text>
    <xsl:value-of select="empty($emptySequence)"/>
    <xsl:text>&#xA;&#xA;      remove($emptySequence, 1) = </xsl:text>
    <xsl:value-of select="remove($emptySequence, 1)"/>
```

```
    <xsl:text>&#xA;&#xA;      empty(remove($emptySequence, 1)) </xsl:text>
    <xsl:text>= </xsl:text>
    <xsl:value-of select="empty(remove($emptySequence, 1))"/>

  </xsl:template>

</xsl:stylesheet>
```

Our last set of results are:

```
Here's a final test of the remove() function:

  Our sequence ($emptySequence) is empty:
      empty($emptySequence) = true

      remove($emptySequence, 1) =

      empty(remove($emptySequence, 1)) = true
```

[2.0] replace()

Given an input string, a regular expression, and a replacement string, replaces all matches of the regular expression in the input string with the replacement string.

Syntax

```
xs:string replace($input as xs:string?, $pattern as xs:string,
                  $replacement as xs:string)
xs:string replace($input as xs:string?, $pattern as xs:string,
                  $replacement as xs:string, $flags as xs:string)
```

Inputs

Three strings; the first string is the original string, the second is a regular expression, and the third is a replacement string. There is also an optional fourth string that specifies flags for how the regular expression should be processed.

It is a fatal error if a regular expression matches a zero-length string. See Appendix E for more details.

Outputs

An updated string in which all matches of the regular expression have been replaced with the replacement string.

Here are the details about how `replace( )` actually works:

- If the regular expression doesn't match anything in the input string, the input string is returned unchanged.

- If the regular expression matches two overlapping strings in the input string, only the first match is replaced.
- Regular expression matching does not use collations; the characters' Unicode code points are compared. Cases in which different characters are considered equal in the world's languages are not taken into account.
- The input string and the replacement string can both be zero-length strings. If the input string is a zero-length string, a zero-length string is returned. If the replacement string is zero-length, any matches for the regular expression are deleted. The regular expression cannot be a zero-length string, nor can it match a zero-length string (`matches("", $pattern, $replacement)` can't be true, in other words). Assuming the input string is not of zero length, it *is* acceptable for a captured substring to be of zero length.

  If the expression `matches("", $pattern, $replacement)` is `true`, an error is raised. For example, using the pattern `'.?'` with any input string raises an error because this pattern matches a zero-length string.
- Unlike regular expressions used in the `<xsl:analyze-string>` element, curly braces ({ and }) are not doubled. (Curly braces used inside the `regex` attribute of `<xsl:analyze-string>` must be doubled so they aren't interpreted as attribute value templates.)
- There are special features and requirements for the replacement string:
  - A literal dollar sign (`$`) symbol must be escaped as `\$`, and a literal backslash (\) must be escaped as \\. An error is raised if a dollar sign is not preceded by a backslash or followed by at least one digit (`[0-9]`). An error is also raised if a backslash is not part of a pair of backslashes (\\) or an escaped dollar sign (`\$`).
  - Within the replacement string, you can use the notation `$N` to indicate the portion of the input string that matches the *N*th parenthesized portion of the regular expression. (This capability is similar to the `regex-group( )` function.) Here is a regular expression we'll use in our example; it represents the pattern for a 16-digit credit card number:

    ```
    ([0-9]{4})-([0-9]{4})-([0-9]{4})-([0-9]{4})
    ```

    There are four parenthesized groups here. (The hyphens between the parentheses aren't in any group.) Given the credit card number `1234-5678-9101-1121`, `$1` represents `1234`, `$2` represents `5678`, `$3` represents `9101`, and `$4` represents `1121`.
  - The notation `$0` refers to the entire string matched by the regular expression. In our example, that happens to be the entire input string, but that's not always the case.
  - If a given regular expression group doesn't match anything in the input string, its value is a zero-length string.

— If you use a single-digit `$N` expression and that digit is greater than the number of regular expression groups, a zero-length string is returned. In our example, `$7` would produce a zero-length string.

— Whenever you use the `$N` notation, the processor assumes every consecutive digit after the dollar sign refers to a group in the regular expression. If that's not the case, it uses the largest number of digits possible, and then the rest of the digits are written to the output.

For example, if we use `$417` in our earlier example, this would generate `112117`. `$4` is the longest expression that matches anything, so the remaining digits (`17`) are output along with the match. If our regular expression has 41 groups, this would be interpreted as `$41` followed immediately by a `7`.

- If a regular expression specifies more than one alternative, and more than one of those alternatives match at the same position in the input, the first alternative in the regular expression is the one that's used. Here's an example from the spec:

```
replace("abcd", "(ab)|(a)", "[1=$1][2=$2]") = "[1=ab][2=]cd"
```

The first alternative in the regular expression `(ab)` is considered the match, so `$1` is `ab` and `$2` is a zero-length string. This is true even though the second alternative is shorter than the first.

- Finally, the `$flags` parameter modifies how the regular expression is processed. There are four different flags:

`s`

Regular expressions are evaluated in what the specs refer to as "dot-all" mode. When this flag is used, the dot operator (`.`) matches any character. Under normal processing (without the `s` flag), the dot operator matches any character *except* the newline character (`&#xA`). This flag is useful when you want to match strings that might include a newline character.

`m`

Regular expressions are evaulated in multiline mode. By default, the meta-character (^) matches the start of the entire string, while $ matches the end of the entire string. In multiline mode, ^ matches the start of any line within the string and $ matches the end of any line within the string.

`i`

Regular expressions are evaluated in case-insensitive mode. The regular expression `"a"` matches both `"a"` and `"A"`.

Note that Unicode issues can complicate this greatly. For example, the XQuery 1.0 and XPath 2.0 Functions and Operators spec gives the example of the Unicode sign for degrees Kelvin (`K`), which is the letter `"K"`. The combination of `regex="k"` and `flags="i"` matches the Kelvin sign as well as the letters `"k"` (`k`) and `"K"` (`K`).

Other Unicode characters don't convert to letters. For example, the Unicode symbol for the Roman numeral I (ࡰ) looks like the letter I, but does not convert to one. Fortunately, these complications are beyond the scope of this book.

x

All whitespace characters (	,
,
, and) are removed from the regular expression before any comparison is done. In other words, with the x flag, the regular expressions "John Smith" and "JohnSmith" are the same. This flag is useful when you want to break a long regular expression into multiple lines to make it easier to read.

The flags can be combined in any order. The parameters 'xis' and 'six' work exactly the same way.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 7.6, "String Functions that use Pattern Matching."

Example

Here is a stylesheet with several tests of the replace( ) function:

```
<?xml version="1.0"?>
<!-- replace.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:variable name="string1"
      select="concat('Now is the time for all good men and ',
              'women to aid the party.')"/>
    <xsl:variable name="string2" as="xs:string"
      select="'Visa # 1234-5678-9101-1121'"/>

    <xsl:text>&#xA;Here's a test of the replace() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>&#xA;  $string1 = &#xA;    </xsl:text>
    <xsl:value-of select="$string1"/>

    <xsl:text>&#xA;&#xA;  Test 1. replace($string1, </xsl:text>
    <xsl:text>'men', 'boys') = &#xA;    </xsl:text>
    <xsl:value-of select="replace($string1, 'men', 'boys')"/>

    <xsl:text>&#xA;&#xA;  Test 2. replace($string1, </xsl:text>
    <xsl:text>' men', ' boys') = &#xA;    </xsl:text>
    <xsl:value-of select="replace($string1, ' men', 'boys')"/>
```

```
    <xsl:text>&#xA;&#xA;  Test 3. replace($string1, </xsl:text>
    <xsl:text>'wombats', 'weasels') = &#xA;    </xsl:text>
    <xsl:value-of select="replace($string1, 'wombats', 'weasels')"/>

    <xsl:text>&#xA;&#xA;  $string2 = &#xA;    </xsl:text>
    <xsl:value-of select="$string2"/>

    <xsl:text>&#xA;&#xA;  Test 4. replace($string2, </xsl:text>
    <xsl:text>&#xA;                  </xsl:text>
    <xsl:text>'([0-9]{4}-)([0-9]{4}-)([0-9]{4}-)([0-9]{4})', </xsl:text>
    <xsl:text>&#xA;                  </xsl:text>
    <xsl:text>'XXXX-XXXX-XXXX-$4') = &#xA;    </xsl:text>
    <xsl:value-of
      select="replace($string2,
              '([0-9]{4})-([0-9]{4})-([0-9]{4})-([0-9]{4})',
              'XXXX-XXXX-XXXX-$4')"/>

    <xsl:text>&#xA;&#xA;  Test 5. replace($string2, </xsl:text>
    <xsl:text>&#xA;                  </xsl:text>
    <xsl:text>'([0-9]{4}-)([0-9]{4}-)([0-9]{4}-)([0-9]{4})', </xsl:text>
    <xsl:text>&#xA;                  </xsl:text>
    <xsl:text>'$0 -> XXXX-XXXX-XXXX-$4') = &#xA;    </xsl:text>
    <xsl:value-of
      select="replace($string2,
              '([0-9]{4})-([0-9]{4})-([0-9]{4})-([0-9]{4})',
              '$0 -> XXXX-XXXX-XXXX-$4')"/>

  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Here's a test of the replace() function:

  $string1 =
    Now is the time for all good men and women to aid the party.

  Test 1. replace($string1, 'men', 'boys') =
    Now is the time for all good boys and woboys to aid the party.

  Test 2. replace($string1, ' men', ' boys') =
    Now is the time for all good boys and women to aid the party.

  Test 3. replace($string1, 'wombats', 'weasels') =
    Now is the time for all good men and women to aid the party.

  $string2 =
    Visa # 1234-5678-9101-1121

  Test 4. replace($string2,
                  '([0-9]{4})-([0-9]{4})-([0-9]{4})-([0-9]{4})',
                  'XXXX-XXXX-XXXX-$4') =
    Visa # XXXX-XXXX-XXXX-1121

  Test 5. replace($string2,
```

```
                '([0-9]{4})-([0-9]{4})-([0-9]{4})-([0-9]{4})',
                '$0 -> XXXX-XXXX-XXXX-$4') =
    Visa # 1234-5678-9101-1121 -> XXXX-XXXX-XXXX-1121
```

Tests 1, 2, and 3 are pretty straightforward. Test 1 replaces text in the input string, but it doesn't work exactly the way we want. Test 2 uses a slightly modified pattern and replacement string to get more useful results. Test 3 returns the input string without any changes because the pattern does not match.

Tests 4 and 5 use the substring feature to display matching text from the original string. In both cases, we're using `replace( )` to hide all but the last four digits of the credit card number. Both tests use the fourth matching substring (`$4`), while test 5 also uses `$0` to display the entire matching string.

See Also

Appendix E has complete details on the way regular expressions work in XPath 2.0. Also see the definitions of the following elements and functions: [2.0] `<xsl:analyze-string>`, [2.0] `matches( )`, [2.0] `<xsl:matching-substring>`, [2.0] `<xsl:non-matching-substring>`, [2.0] `regex-group( )`, and [2.0] `tokenize( )`.

[2.0] resolve-QName()

This is a function that constructs a `QName`. Given an `xs:string` in the format of a (optionally prefixed) name and an element, this function returns a `QName`. The prefix of the name, if any, is resolved using the namespaces in scope for the given element.

Syntax

```
xs:QName? resolve-QName($qname as xs:string?, $element as element())
```

Inputs

An `xs:string` that is a valid XML name (it can contain a namespace prefix as well) and an element.

Outputs

A new `xs:QName` value.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 11.1, "Additional Constructor Functions for QNames."

Example

To create a new `QName` value with `resolve-QName( )`, we'll revisit our Shakespearean sonnet:

```
<?xml version="1.0"?>
<!-- sonnet.xml -->
<sonnet type='Shakespearean'>
  <auth:author xmlns:auth="http://www.authors.com/">
```

```
    <last-name>Shakespeare</last-name>
    <first-name>William</first-name>
    <nationality>British</nationality>
    <year-of-birth>1564</year-of-birth>
    <year-of-death>1616</year-of-death>
  </auth:author>
  <!-- Is there an official title for this sonnet?  They're
       sometimes named after the first line.                    -->
  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
    <line>If hairs be wires, black wires grow on her head.</line>
    <line>I have seen roses damasked, red and white,</line>
    <line>But no such roses see I in her cheeks.</line>
    <line>And in some perfumes is there more delight</line>
    <line>Than in the breath that from my mistress reeks.</line>
    <line>I love to hear her speak, yet well I know</line>
    <line>That music hath a far more pleasing sound.</line>
    <line>I grant I never saw a goddess go,</line>
    <line>My mistress when she walks, treads on the ground.</line>
    <line>And yet, by Heaven, I think my love as rare</line>
    <line>As any she belied with false compare.</line>
  </lines>
</sonnet>
```

Here is our stylesheet:

```
<?xml version="1.0"?>
<!-- resolve-qname.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the resolve-QName() function:&#xA;</xsl:text>

    <xsl:variable name="testQName1" as="xs:QName"
      select="resolve-QName('auth:last-name',
              /sonnet/*:author/last-name)"/>

    <xsl:text>&#xA;  testQName1 = resolve-QName('auth:</xsl:text>
    <xsl:text>last-name', /sonnet/author/last-name)</xsl:text>
    <xsl:text>&#xA;  prefix-from-QName</xsl:text>
    <xsl:text>($testQName1) = "</xsl:text>
    <xsl:value-of select="prefix-from-QName($testQName1)"/>
    <xsl:text>"&#xA;  local-name-from-QName($testQName1) = "</xsl:text>
    <xsl:value-of select="local-name-from-QName($testQName1)"/>
    <xsl:text>"&#xA;  namespace-uri-from-QName($testQName1) = "</xsl:text>
    <xsl:value-of select="namespace-uri-from-QName($testQName1)"/>
    <xsl:text>"&#xA;</xsl:text>

    <xsl:variable name="testQName2" as="xs:QName"
```

```
      select="resolve-QName('something-else', /sonnet)"/>

    <xsl:text>&#xA;  testQName2 = resolve-QName('something-</xsl:text>
    <xsl:text>else', /sonnet)</xsl:text>
    <xsl:text>&#xA;  prefix-from-QName</xsl:text>
    <xsl:text>($testQName2) = "</xsl:text>
    <xsl:value-of select="prefix-from-QName($testQName2)"/>
    <xsl:text>"&#xA;  local-name-from-QName($testQName2) = "</xsl:text>
    <xsl:value-of select="local-name-from-QName($testQName2)"/>
    <xsl:text>"&#xA;  namespace-uri-from-QName($testQName2) = "</xsl:text>
    <xsl:value-of select="namespace-uri-from-QName($testQName2)"/>
    <xsl:text>"</xsl:text>

    <!-- This raises an error; the 'auth' prefix isn't in scope for -->
    <!-- the <sonnet> element. -->
    <!--    <xsl:variable name="testQName3" as="xs:QName"
              select="resolve-QName('auth:new-element', /sonnet)"/> -->
  </xsl:template>

</xsl:stylesheet>
```

And here are the results:

```
Tests of the resolve-QName() function:

  testQName1 = resolve-QName('auth:last-name', /sonnet/author/last-name)
  prefix-from-QName($testQName1) = "auth"
  local-name-from-QName($testQName1) = "last-name"
  namespace-uri-from-QName($testQName1) = "http://www.authors.com/"

  testQName2 = resolve-QName('something-else', /sonnet)
  prefix-from-QName($testQName2) = ""
  local-name-from-QName($testQName2) = "something-else"
  namespace-uri-from-QName($testQName2) = ""
```

In our first use of `resolve-QName( )`, we created a variable of datatype `xs:QName` that used the `auth` prefix in the scope of the `<last-name>` element. Notice that the `auth` prefix is actually defined on the `<last-name>`'s parent, but it is in scope. Once we've created the `QName`, we use the `*-from-QName( )` functions to display its parts.

The second example here creates a `QName` that doesn't have a namespace. We're using a local name that doesn't have a prefix, so the `prefix-from-QName( )` function returns a zero-length string. The second argument to `resolve-QName( )` is a `<sonnet>` element. Our sonnet doesn't have a default namespace, so our `QName` isn't in a namespace.

Finally, the commented-out code in the stylesheet attempts to use the `auth` namespace at the root element (`/sonnet`). The `auth` namespace is not in scope at the root element, so this call to `resolve-QName( )` raises an error.

[2.0] resolve-uri()

Resolves a relative URI against an absolute URI.

Syntax

```
xs:anyURI? resolve-uri($relative as xs:string?)
xs:anyURI? resolve-uri($relative as xs:string?, $base as xs:string)
```

Inputs

An `xs:string` representing a relative URI. An optional `xs:string` containing an absolute URI is accepted as well. If no absolute URI is given, the base URI from the static context is used.

Outputs

An `xs:anyURI` containing the resolved URI. If the second argument is not provided and no base URI is defined in the current context, an error is raised. If either input string is not a valid URI, an error is raised. If the relative URI is an absolute URI, it is returned unchanged. Finally, if the relative URI is the empty sequence, the empty sequence is returned.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 8, "Functions on `anyURI`."

Example

Here is a stylesheet that resolves two URIs. In one we use the base URI from the static context, while in the other we supply an absolute URI.

```
<?xml version="1.0"?>
<!-- resolve-uri.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the resolve-uri() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  The static base URI for the </xsl:text>
    <xsl:text>document root is:&#xA;    </xsl:text>
    <xsl:value-of select="static-base-uri()"/>

    <xsl:text>&#xA;&#xA;  The base URI for the </xsl:text>
    <xsl:text>document root is:&#xA;    </xsl:text>
    <xsl:value-of select="base-uri()"/>

    <xsl:text>&#xA;&#xA;  resolve-uri('doug.html') = </xsl:text>
    <xsl:text>&#xA;    </xsl:text>
    <xsl:value-of select="resolve-uri('doug.html')"/>

    <xsl:text>&#xA;&#xA;  resolve-uri('doug.html', </xsl:text>
    <xsl:text>'http://www.oreilly.com/') = </xsl:text>
    <xsl:text>&#xA;    </xsl:text>
    <xsl:value-of
      select="resolve-uri('doug.html', 'http://www.oreilly.com/')"/>

  </xsl:template>
```

```
</xsl:stylesheet>
```

(We use the `static-base-uri( )` function to print out the static base URI for the transformation and use the `base-uri( )` function to print out the base URI property before we use the `resolve-uri( )` function.) The results look like this:

```
Tests of the resolve-uri() function:

  The static base URI for the document root is:
    file:/C://resolve-uri.xsl

  The base URI for the document root is:
    file:/C://chocolate.xml

  resolve-uri('doug.html') =
    file:/C:/doug.html

  resolve-uri('doug.html', 'http://www.oreilly.com/') =
    http://www.oreilly.com/doug.html
```

We get these results when invoking this stylesheet against the file *chocolate.xml*. That file's location on the file system becomes the base URI for the document. The first test creates a URI for a document in the same directory as our input document. The second test takes the name of a resource and the URL of a web site and combines them to create a URI.

[2.0] reverse()

Given a sequence of items, returns a sequence with the items in reverse order.

Syntax

```
item()* reverse(item()*)
```

Inputs

A sequence of items. If the sequence is the empty sequence, the empty sequence is returned.

Outputs

A sequence with the input items in reverse order.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.1, "General Functions and Operators on Sequences."

Example

Here is a short stylesheet that creates a sequence and prints it, and then prints the sequence again after invoking the `reverse( )` function against it:

```
<?xml version="1.0"?>
<!-- reverse.xsl -->
<xsl:stylesheet version="2.0"
```

```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:datatest="http://www.oreilly.com">

    <xsl:output method="text"/>

    <xsl:template match="/">

      <xsl:variable name="testSequence" as="item()*">
        <xsl:sequence
          select="(3, 4, 5, current-date(), current-time(), 8, 'blue',
                   'red', xs:float(3.14), 42, xs:date('1995-04-21'))"/>
      </xsl:variable>

      <xsl:text>&#xA;Here is a test of the reverse() </xsl:text>
      <xsl:text>function:&#xA;</xsl:text>

      <xsl:text>&#xA;  Our original sequence is:&#xA;&#xA;    </xsl:text>
      <xsl:value-of select="$testSequence" separator="&#xA;    "/>

      <xsl:text>&#xA;&#xA;  Passing our sequence to reverse() </xsl:text>
      <xsl:text>gives us:&#xA;&#xA;    </xsl:text>

      <xsl:value-of select="reverse($testSequence)" separator="&#xA;    "/>
    </xsl:template>

  </xsl:stylesheet>
```

Here are the results:

```
  Here is a test of the reverse() function:

    Our original sequence is:

      3
      4
      5
      2006-07-22-04:00
      13:48:23.256-04:00
      8
      blue
      red
      3.14
      42
      1995-04-21

    Passing our sequence to reverse() gives us:

      1995-04-21
      42
      3.14
      red
      blue
      8
      13:48:23.256-04:00
      2006-07-22-04:00
```

```
5
4
3
```

Remember that variables don't change, so the original sequence still exists. If we want to use the reversed sequence, we would have to store it in a new variable.

[2.0] root()

Given a node, returns the root of the tree to which the node belongs.

Syntax

```
node() root()
node() root(node()?)
```

Inputs

An optional node. If no node is provided, the context node is used.

Outputs

The root of the tree to which the node (or the context node, if the call was `root( )`) belongs. This will usually be a document node, but that isn't always the case.

If an argument is provided, and that argument is the empty sequence, the empty sequence is returned. Also, if the argument itself is a document node, the argument is simply returned.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 14, "Functions and Operators on Nodes."

Example

To test the `root( )` function, we'll take another look at our list of favorite albums:

```
<?xml version="1.0"?>
<!-- favorites.xml -->
<list>
  <title>A few of my favorite albums</title>
  <listitem>A Love Supreme</listitem>
  <listitem>Beat Crazy</listitem>
  <listitem>Here Come the Warm Jets</listitem>
  <listitem>Kind of Blue</listitem>
  <listitem>London Calling</listitem>
  <listitem>Remain in Light</listitem>
  <listitem>The Joshua Tree</listitem>
  <listitem>The Indestructible Beat of Soweto</listitem>
</list>
```

We'll use this stylesheet to test the `root( )` function:

```
<?xml version="1.0"?>
<!-- root.xsl -->
```

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:ora="http://www.oreilly.com">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the root() function:</xsl:text>

    <xsl:variable name="testTree" as="node()*">
      <report month="8" year="2006">
        <title>Chocolate bar sales</title>
        <brand>
          <name>Lindt</name>
          <units>27408</units>
        </brand>
      </report>
      <otherReport month="9" year="2006">
        <title>Chocolate bar sales</title>
        <brand>
          <name>Callebaut</name>
          <units>8203</units>
        </brand>
      </otherReport>
    </xsl:variable>

    <xsl:text>&#xA;&#xA;  1. root(//listitem[3]):</xsl:text>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of
      select="ora:doc-node-test(root(//listitem[3]))"/>

    <xsl:text>&#xA;&#xA;  2. root(subsequence($testTree, 1, 1)):</xsl:text>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of
      select="ora:doc-node-test(root(subsequence($testTree, 1, 1)))"/>

    <xsl:text>&#xA;&#xA;  3. root(subsequence($testTree, 2)):</xsl:text>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of
      select="ora:doc-node-test(root(subsequence($testTree, 2, 1)))"/>

    <xsl:text>&#xA;&#xA;  4. root():</xsl:text>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of
      select="ora:doc-node-test(root())"/>

  </xsl:template>

  <xsl:function name="ora:doc-node-test" as="xs:string">
    <xsl:param name="node" as="node()*"/>
    <xsl:choose>
      <xsl:when test="empty($node)">
        <xsl:value-of select="'       [empty sequence]'"/>
      </xsl:when>
```

```
    <xsl:when test="$node instance of document-node()">
      <xsl:value-of
        select="concat('       This IS a document node.&#xA;',
                '       root node''s name: &lt;',
                node-name($node/*[1]),
                '&gt;')"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of
        select="concat('       This IS NOT a document node.&#xA;',
                '       root node''s name: &lt;',
                node-name($node),
                '&gt;')"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:function>

</xsl:stylesheet>
```

Notice that we define a variable named `testTree` that contains two elements; you can't have two root elements in an XML document, but it is legal in a variable. We'll use this variable to demonstrate calls to `root( )` that don't return a document node.

Our stylesheet uses an `<xsl:function>` to refactor out the code that processes the various root elements. Given a sequence, our function `ora:doc-node-test( )` first checks to see whether the sequence is the empty sequence. (For example, if we use this stylesheet to transform an XML document that doesn't contain three `<listitem>` elements, the sequence is empty.)

Assuming the sequence is not empty, we use the `document-node( )` node test to see whether it is a document node. If it is, we print the name of its first child element. A document node can technically contain more than one child, so we use the node test `$node/*[1]` to select its first child node. (See the discussion of the [2.0] `<xsl:document>` element in Appendix A for an example of a document node with more than one child element.) If this is not a document node, we simply pass the node itself to the `node-name( )` function.

To make our stylesheet more robust, we could write the XPath expression like this:

```
<xsl:when test="($node treat as node()) instance of document-node()">
```

The variable `$node` has a type of `node( )*`; the `treat as` instruction casts it to a type of `node( )`. We could also use the `exactly-one( )` function to ensure the sequence has only one item. Our stylesheet makes sure that we only call the `ora:doc-node-test( )` function with a singleton.

Now that we've discussed the function, we'll look at the four tests of the `root( )` function. The first use of `root( )` gets the root item of the third `<listitem>` element in the context item. The second and third calls to `root( )` use the two root nodes in the variable `$testTree`. While both of them have root nodes, neither root node is a document node. For the last use of `root( )`, we send the root of the context item, which is the root node of the XML document we're transforming.

Here are the results we get:

```
Tests of the root() function:

  1. root(//listitem[3]):
       This IS a document node.
       root node's name: <list>

  2. root(subsequence($testTree, 1, 1)):
       This IS NOT a document node.
       root node's name: <report>

  3. root(subsequence($testTree, 2)):
       This IS NOT a document node.
       root node's name: <otherReport>

  4. root():
       This IS a document node.
       root node's name: <list>
```

round()

Returns the integer closest to the argument.

Syntax

[1.0] number? **round(***number?***)**
[2.0] numeric? **round(***numeric?***)**

Inputs

[1.0] A number. If the argument is not a number, it is converted to a number as if it were passed to the `number( )` function.

[2.0] In XSLT 2.0, the argument for `round( )` must be a number (`xs:integer`, `xs:float`, `xs:decimal`, or `xs:double`, or a datatype derived from them). If the argument is not a number, the XSLT processor raises an error. For example, if the argument is an untyped value read in from an XML document that doesn't use a schema, it must be cast to a numeric value before it is passed to `round( )`.

Output

The integer that is closest to the argument. If two numbers are equally close to the argument (`1` and `2` are equally close to `1.5`), the number closest to positive infinity is returned. Various argument values are handled as follows:

- If the argument is positive infinity, then positive infinity is returned.
- If the argument is negative infinity, then negative infinity is returned.
- If the argument is positive zero, then positive zero is returned.
- If the argument is negative zero, then negative zero is returned.
- If the argument is between zero and `-0.5`, then negative zero is returned.

- If the argument is `NaN` (not a number), the `round( )` function returns `NaN`.

[2.0] In XSLT 2.0, the output value has the same datatype as the input value. In other words, the function `round(xs:integer)` returns an `xs:integer`, while `round(xs:float)` returns an `xs:float`.

Defined in

[1.0] XPath section 4.4, "Number Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 6.4, "Functions on Numeric Values."

Example

The following stylesheet shows the results of invoking the `round( )` function against a variety of values. We'll use this XML document as input:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

Here's the stylesheet that uses the `round( )` function:

```
<?xml version="1.0"?>
<!-- round1.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the round() function:&#xA;&#xA;</xsl:text>
```

```
    <xsl:text>  round(7.983) = </xsl:text>
    <xsl:value-of select="round(7.983)"/>

    <xsl:text>&#xA;  round(-7.893) = </xsl:text>
    <xsl:value-of select="round(-7.893)"/>

    <xsl:text>&#xA;  round(avg(/report/brand/units)) = </xsl:text>
    <xsl:value-of select="round(avg(/report/brand/units))"/>

    <xsl:text>&#xA;  round('blue') = </xsl:text>
    <xsl:value-of select="round('blue')"/>

  </xsl:template>

</xsl:stylesheet>
```

When we process our XML document with this stylesheet, the results are:

```
Tests of the round() function:

  round(7.983) = 8
  round(-7.893) = -8
  round(avg(/report/brand/units)) = 18263
  round('blue') = NaN
```

If we change the `<xsl:stylesheet>` element to have `version="2.0"`, the stylesheet won't run at all because `round('blue')` is a static error. We can add `version="1.0"` to the `<xsl:value-of>` element so the element is processed in XSLT 1.0 mode. We can also use the `number( )` function to convert `blue` to a number. Here's how the stylesheet looks:

```
<?xml version="1.0"?>
<!-- round2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    ...
    <xsl:text>&#xA;  [1.0] round('blue') = </xsl:text>
            <xsl:value-of version="1.0" select="round('blue')"/>

    <xsl:text>&#xA;  round(number('blue')) = </xsl:text>
    <xsl:value-of select="round(number('blue'))"/>
  </xsl:template>

</xsl:stylesheet>
```

The results are similar:

```
Tests of the round() function:
  ...
  [1.0] round('blue') = NaN
  round(number('blue')) = NaN
```

[2.0] round-half-to-even()

Returns the integer closest to the argument, with the exception that any value that ends with `.5` is rounded to the nearest *even* number.

Syntax

```
numeric? round-half-to-even(numeric?)
numeric? round-half-to-even(numeric?, $precision as xs:integer)
```

Inputs

A numeric value and an optional `xs:integer` specifying the number of digits of precision to be used in the calculation. The numeric value must be of type `xs:float`, `xs:double`, `xs:decimal`, or `xs:integer`. If it is not, the XSLT processor raises an error.

Output

The integer closest to the argument, with values ending in `.5` rounded to the nearest even number. With that exception, `round-half-to-even( )` works the same as `round( )`:

- If the argument is positive infinity, then positive infinity is returned.
- If the argument is negative infinity, then negative infinity is returned.
- If the argument is positive zero, then positive zero is returned.
- If the argument is negative zero, then negative zero is returned.
- If the argument is between zero and `-0.5`, then negative zero is returned.
- If the argument is `NaN` (not a number), then `NaN` is returned.

If the `$precision` argument is used, the XSLT processor returns the number closest to the value that is a multiple of 10 to the power of *minus* `$precision`. The value `-2` rounds the number to the nearest `100`, while the value `2` rounds the number to the nearest `.01`.

The output value has the same datatype as the input value. In other words, the function `round-half-to-even(xs:integer)` returns an `xs:integer`, while `round-half-to-even(xs:float)` returns an `xs:float`.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 6.4, "Functions on Numeric Values."

Example

Here is a simple stylesheet that illustrates how `round-half-to-even( )` works:

```
<?xml version="1.0"?>
<!-- round-half-to-even.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>
```

```
  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the round-half-to-even() function:&#xA;</xsl:text>

    <xsl:text>&#xA;  round-half-to-even(-0.5) = </xsl:text>
    <xsl:value-of select="round-half-to-even(-0.5)"/>

    <xsl:text>&#xA;  round-half-to-even(0.5) = </xsl:text>
    <xsl:value-of select="round-half-to-even(0.5)"/>

    <xsl:text>&#xA;  round-half-to-even(1.5) = </xsl:text>
    <xsl:value-of select="round-half-to-even(1.5)"/>

    <xsl:text>&#xA;  round-half-to-even(2.5) = </xsl:text>
    <xsl:value-of select="round-half-to-even(2.5)"/>

    <xsl:text>&#xA;  round-half-to-even(number('NaN')) = </xsl:text>
    <xsl:value-of select="round-half-to-even(number('NaN'))"/>

    <xsl:text>&#xA;  round-half-to-even</xsl:text>
    <xsl:text>(avg(/report/brand/units)) = </xsl:text>
    <xsl:value-of select="round-half-to-even(avg(/report/brand/units))"/>

    <xsl:text>&#xA;  round-half-to-even</xsl:text>
    <xsl:text>(avg(/report/brand/units), -2) = </xsl:text>
    <xsl:value-of
      select="round-half-to-even(avg(/report/brand/units), -2)"/>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet generates these results:

```
Tests of the round-half-to-even() function:

  round-half-to-even(-0.5) = 0
  round-half-to-even(0.5) = 0
  round-half-to-even(1.5) = 2
  round-half-to-even(2.5) = 2
  round-half-to-even(avg(/report/brand/units)) = 18263
  round-half-to-even(avg(/report/brand/units), -2) = 18300
```

Notice the effect of the `$precision` argument in the last line. The average sales figure has been rounded to the nearest multiple of 100 (10 to the power of minus `-2`).

[2.0] seconds-from-dateTime()

Given an `xs:dateTime` value, returns its seconds value.

Syntax

```
xs:decimal? seconds-from-dateTime(xs:dateTime?)
```

Inputs

An `xs:dateTime` value.

Output

An `xs:decimal` representing the seconds component of the given `xs:dateTime` value. If the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `seconds-from-dateTime( )` function:

```
<?xml version="1.0"?>
<!-- seconds-from-datetime.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the seconds from an xs:dateTime:</xsl:text>
    <xsl:variable name="currentDateTime" as="xs:dateTime"
      select="current-dateTime()"/>
    <xsl:text>&#xA;&#xA;  The current date and time is: </xsl:text>
    <xsl:value-of select="$currentDateTime"/>

    <xsl:text>&#xA;&#xA;  The current seconds are: </xsl:text>
    <xsl:value-of select="seconds-from-dateTime($currentDateTime)"/>
    <xsl:text>, &#xA;    usually written as a whole number (</xsl:text>
    <xsl:value-of select="format-dateTime($currentDateTime, '[s01]')"/>
    <xsl:text>)</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet creates these results:

```
Extracting the seconds from an xs:dateTime:

  The current date and time is: 2006-11-16T05:02:55.438-05:00

  The current seconds are: 55.438,
    usually written as a whole number (55)
```

See Also

The definitions of the [2.0] `day-from-dateTime( )`, [2.0] `format-dateTime( )`, [2.0] `hours-from-dateTime( )`, [2.0] `minutes-from-dateTime( )`, [2.0] `month-from-dateTime( )`, [2.0] `timezone-from-dateTime( )`, and [2.0] `year-from-dateTime( )` functions.

[2.0] seconds-from-duration()

Given an `xs:duration` value, returns the number of seconds in that duration.

Syntax

```
xs:decimal? seconds-from-duration(xs:duration?)
```

Inputs

An `xs:duration` value.

Output

An `xs:decimal` representing the seconds component of the given `xs:duration`. Be aware that for an `xs:yearMonthDuration`, this function always returns `0` because there is no seconds component of an `xs:yearMonthDuration`. Also, if the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `seconds-from-duration( )` function with all three types of durations:

```
<?xml version="1.0"?>
<!-- seconds-from-duration.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the seconds component from durations:</xsl:text>

    <xsl:variable name="sampleDuration" as="xs:duration"
      select="xs:duration('P3Y8M2DT4H23M12.2S')"/>
    <xsl:variable name="sampleYearMonthDuration" as="xs:yearMonthDuration"
      select="xs:yearMonthDuration('P3Y8M')"/>
    <xsl:variable name="sampleDayTimeDuration" as="xs:dayTimeDuration"
      select="xs:dayTimeDuration('P2DT4H23M12.2S')"/>

    <xsl:text>&#xA;&#xA;  A sample xs:duration: </xsl:text>
    <xsl:value-of select="$sampleDuration"/>
    <xsl:text>&#xA;    The seconds component of this duration is </xsl:text>
    <xsl:value-of select="seconds-from-duration($sampleDuration)"/>
    <xsl:text>.</xsl:text>

    <xsl:text>&#xA;&#xA;  A sample xs:yearMonthDuration: </xsl:text>
    <xsl:value-of select="$sampleYearMonthDuration"/>
```

```
    <xsl:text>&#xA;    The seconds component of this duration is </xsl:text>
    <xsl:value-of select="seconds-from-duration($sampleYearMonthDuration)"/>
    <xsl:text>.</xsl:text>

    <xsl:text>&#xA;&#xA;  A sample xs:dayTimeDuration: </xsl:text>
    <xsl:value-of select="$sampleDayTimeDuration"/>
    <xsl:text>&#xA;    The seconds component of this duration is </xsl:text>
    <xsl:value-of select="seconds-from-duration($sampleDayTimeDuration)"/>
    <xsl:text>.</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates these results:

```
Extracting the seconds component from durations:

  A sample xs:duration: P3Y8M2DT4H23M12.2S
    The seconds component of this duration is 12.2.

  A sample xs:yearMonthDuration: P3Y8M
    The seconds component of this duration is 0.

  A sample xs:dayTimeDuration: P2DT4H23M12.2S
    The seconds component of this duration is 12.2.
```

As we mentioned earlier, extracting the seconds component from an `xs:yearMonthDuration` returns `0`.

See Also

The definitions of the [2.0] `days-from-duration( )`, [2.0] `hours-from-duration( )`, [2.0] `minutes-from-duration( )`, [2.0] `months-from-duration( )`, and [2.0] `years-from-duration( )` functions.

[2.0] seconds-from-time()

Given an `xs:time` value, returns its seconds component.

Syntax

```
xs:decimal? seconds-from-time(xs:time?)
```

Input

An `xs:time` value.

Output

An `xs:decimal` representing the seconds component of the given `xs:time` value. If the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

Here is a short stylesheet that uses the `seconds-from-time( )` function:

```
<?xml version="1.0"?>
<!-- seconds-from-time.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the seconds component from an xs:time:</xsl:text>
    <xsl:variable name="currentTime" as="xs:time" select="current-time()"/>
    <xsl:text>&#xA;&#xA;  The current time is: </xsl:text>
    <xsl:value-of select="$currentTime"/>

    <xsl:text>&#xA;&#xA;  The current seconds are: </xsl:text>
    <xsl:value-of select="seconds-from-time($currentTime)"/>
    <xsl:text>, &#xA;    usually written as a whole number (</xsl:text>
    <xsl:value-of select="format-time($currentTime, '[s01]')"/>
    <xsl:text>)</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet generates these results:

```
Extracting the seconds component from an xs:time:

  The current time is: 05:07:40.307-05:00

  The current seconds are: 40.307,
    usually written as a whole number (40)
```

Notice that the first result was generated by the `seconds-from-time( )` function, while the second was generated by `format-time( )`.

See Also

The definitions of the [2.0] `hours-from-time( )`, [2.0] `format-time( )`, [2.0] `minutes-from-time( )`, [2.0] `seconds-from-time( )`, and [2.0] `timezone-from-time( )` functions.

starts-with()

Determines whether the first argument string begins with the second argument.

Syntax

```
[1.0] boolean starts-with(string, string)
[2.0] xs:boolean starts-with(xs:string, xs:string)
[2.0] xs:boolean starts-with(xs:string, xs:string, $collation as xs:string)
```

Inputs

Two strings.

[2.0] In XSLT 2.0, there is an optional third argument—the name of a collation that specifies how strings are compared.

Output

Assuming both arguments are not zero-length strings, if the first string begins with the second, `starts-with( )` returns the boolean value `true`; otherwise, it returns `false`. If the second string is a zero-length string, `starts-with( )` returns `true`. If the first string *and* the second string are both zero-length strings, `starts-with( )` returns `true`. If the first string is a zero-length string but the second string is not, `starts-with( )` returns `false`.

Defined in

[1.0] XPath section 4.2, "String Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 7.5, "Functions Based on Substring Matching."

Example

We'll use this sample XML document:

```
<?xml version="1.0"?>
<!-- favorites.xml -->
<list>
  <title>A few of my favorite albums</title>
  <listitem>A Love Supreme</listitem>
  <listitem>Beat Crazy</listitem>
  <listitem>Here Come the Warm Jets</listitem>
  <listitem>Kind of Blue</listitem>
  <listitem>London Calling</listitem>
  <listitem>Remain in Light</listitem>
  <listitem>The Joshua Tree</listitem>
  <listitem>The Indestructible Beat of Soweto</listitem>
</list>
```

This stylesheet outputs the contents of all `<listitem>` elements that begin with the string "The":

```
<?xml version="1.0"?>
<!-- starts-with.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
```

```
    <xsl:text>&#xA;A test of the starts-with() </xsl:text>
    <xsl:text>function:&#xA;&#xA;</xsl:text>
    <xsl:for-each select="list/listitem[starts-with(., 'The')]">
      <xsl:number count="listitem" format="1. "/>
      <xsl:value-of select="."/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Our stylesheet generates these results:

```
A test of the starts-with() function:

7. The Joshua Tree
8. The Indestructible Beat of Soweto
```

We used `starts-with( )` inside an XPath predicate expression to select all of the `<listitem>` nodes that begin with `The`.

The `<xsl:number>` element indicates the position of each item in the original list. Using `position( )` to generate item numbers labels the items as `1` and `2` because these are the only two nodes being processed inside the `<xsl:for-each>` element.

[2.0] The `starts-with` function is one of several that can contain an optional collation. How that collation is specified varies from one processor to the next. In Saxon this is done with a URI that includes the name of the Java class that performs the collation. Here's an example that compares the German words *Straße* and *Strasse*. In the default collation, these two words are different; using the German collation, they're the same. Here's the stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- starts-with2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:variable name="string1" select="'Stra&#xDF;e'"/>
    <xsl:variable name="string2" select="'Strasse'"/>

    <xsl:text>&#xA;A test of the starts-with() function:&#xA;</xsl:text>

    <xsl:text>  starts-with('</xsl:text>
    <xsl:value-of select="$string1"/>
    <xsl:text>', '</xsl:text>
    <xsl:value-of select="$string2"/>
    <xsl:text>') = </xsl:text>
    <xsl:value-of select="starts-with($string1, $string2)"/>
    <xsl:text>&#xA;</xsl:text>

    <xsl:text>  starts-with('</xsl:text>
    <xsl:value-of select="$string1"/>
    <xsl:text>', '</xsl:text>
    <xsl:value-of select="$string2"/>
```

```
    <xsl:text>', [German collation]) = </xsl:text>
    <xsl:value-of
      select="starts-with($string1, $string2,
              concat('http://saxon.sf.net/collation?',
                     'class=com.oreilly.xslt.GermanCollation;'))"/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

And here are the results:

```
A test of the starts-with() function:
  starts-with('Straße', 'Strasse') = false
  starts-with('Straße', 'Strasse', [German collation]) = true
```

Using the collation we defined, the two spellings of `Strasse` are the same; using the default collation, they're not. (To keep the listing within the margins of the page, we used the `concat( )` function to combine the two halves of the Saxon collation URI.) See "The document() Function and Sorting" in Chapter 8 for more information.

[2.0] static-base-uri()

Returns the value of the base URI property from the static context. The base URI property can change from one XML element to the next, but `static-base-uri( )` returns the same value regardless of the location in the input document. In Saxon-J 9.0.0.3, the static base URI is the base URI of the stylesheet. Altova XML works similarly, although the format of the URI is not exactly the same.

Syntax

```
xs:anyURI? static-base-uri()
```

Inputs

None.

Outputs

An `xs:anyURI` that represents the static base URI. If the base URI property is not defined in the static context, `static-base-uri( )` returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 16, "Context Functions."

Example

To illustrate how `static-base-uri( )` works, we'll reuse our earlier document that used the `xml:base` attribute to define different base URIs throughout the document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- xmlbase.xml -->
<cars>
```

```
  <manufacturer name="Chevrolet"
    xml:base="http://www.chevrolet.com/">
    <car>Cavalier</car>
    <car>Corvette</car>
    <car>Impala</car>
    <car>Malibu</car>
  </manufacturer>
  <manufacturer name="Ford"
    xml:base="http://www.ford.com/">
    <car>Pinto</car>
    <car>Mustang</car>
    <car>Taurus</car>
  </manufacturer>
  <manufacturer name="Volkswagen"
    xml:base="http://www.vw.com/">
    <car>Beetle</car>
    <car>Jetta</car>
    <car>Passat</car>
    <car>Touraeg</car>
  </manufacturer>
</cars>
```

Here's our stylesheet. We call the `static-base-uri( )` function at the document root, and we also call it for the `<manufacturer>` elements that change the base URI:

```
<?xml version="1.0"?>
<!-- static-base-uri.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the static-base-uri() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  The static base URI for the </xsl:text>
    <xsl:text>document root is:&#xA;    </xsl:text>
    <xsl:value-of select="static-base-uri()"/>
    <xsl:text>&#xA;&#xA;</xsl:text>

    <xsl:for-each select="/cars/manufacturer">
      <xsl:text>  The static base URI for manufacturer </xsl:text>
      <xsl:value-of select="@name"/>
      <xsl:text> is: &#xA;    </xsl:text>
      <xsl:value-of select="static-base-uri()"/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Here are our results:

```
Tests of the static-base-uri() function:

  The static base URI for the document root is:
    file:/C://static-base-uri.xsl
```

```
The static base URI for manufacturer Chevrolet is:
  file:/C://static-base-uri.xsl
The static base URI for manufacturer Ford is:
  file:/C://static-base-uri.xsl
The static base URI for manufacturer Volkswagen is:
  file:/C://static-base-uri.xsl
```

As you can see, the `static-base-uri( )` function always returns the same value (the URI of the stylesheet), regardless of where we are in the XML document or whether any `xml:base` attributes have been defined.

string()

Returns the string value of the argument.

Syntax

```
[1.0] string string(object)
[2.0] xs:string string(item()?)
```

Inputs

An *[1.0]* object or *[2.0]* item. The argument is converted to a string as described in the following subsection.

Output

[1.0] A string. The input argument is converted to a string as follows:

- If the argument is a node-set, the first node in the node-set is converted to a string. (The first node in the node-set is the one that occurs first in document order.) For an element or root node, this means all of its text content. For an attribute, this means its value.
- If the argument is a number, it is converted to a string as follows:
  - The value `NaN` is converted to the string `"NaN"`.
  - Positive zero is converted to the string `"0"`.
  - Negative zero is converted to the string `"0"`.
  - Positive infinity is converted to the string `"Infinity"`.
  - Negative infinity is converted to the string `"-Infinity"`.
  - An integer is converted to a string representing that integer, using no decimal point and no leading zeros. If the integer is negative, it will be preceded by a minus sign (-).
  - Any other number is converted to a string with a decimal point—at least one number before the decimal point and at least one number after the decimal point. If the number is negative, it will be preceded by a minus sign (-). There will not be any leading zeros before the decimal point (with the possible

exception of the one required digit before the decimal point). After the decimal point, there will be only as many digits as needed to distinguish this number from all other numeric values defined by the IEEE 754 standard, the same standard used by the Java `float` and `double` types.

- If the argument is a boolean value, the value `true` is represented by the string `"true"` and the value `false` is represented by the string `"false"`.
- If the argument is any other type, it is converted to a string in a way that depends on that type. See the documentation for your XSLT processor to find out what other types are supported and how they are converted to strings.

[2.0] An `xs:string`. Because XSLT 2.0 supports more datatypes than XSLT 1.0, it's not surprising that the rules for converting values to strings are more complicated. The rules are:

- If the item is an `xs:string` or any datatype derived from `xs:string`, the argument is returned without any changes.
- If the item is a sequence, it must be either the empty sequence or a singleton. *It is an error to call* `string( )` *with a sequence containing more than one item.* This is a major change from XSLT 1.0, in which `string( )` (and a number of other functions) simply took the first item in the node-set and processed it. If you are migrating an XSLT 1.0 stylesheet to XSLT 2.0, you'll have to change this code:

  ```
  <xsl:value-of select="string(items/item/price)"/>
  ```

  to this:

  ```
  <xsl:value-of select="string((items/item/price)[1])"/>
  ```

  If the `string( )` function doesn't have an argument, the context item is used. In other words, `string( )` is the same as `string(.)`. Calling `string( )` with the empty sequence returns a zero-length string.
- If the item is numeric (`xs:integer`, `xs:decimal`, `xs:float`, or `xs:double`, or anything derived from them), it is converted to a string using these rules:

  —If the value is `NaN` (not a number), it is converted to the string `"NaN"`.

  —If the value is positive infinity, it is converted to the string `"INF"`.

  —If the value is negative infinity, it is converted to the string `"-INF"`.

  —If the value is positive zero, it is converted to the string `0`.

  —If the value is negative zero, it is converted to the string `-0`.

  The XQuery 1.0 and XPath 2.0 Functions and Operators spec goes into exhaustive detail about how various numeric datatypes are converted to strings. For example, an `xs:integer` value will not have a decimal point when it is converted to a string. There are also rules for how a mantissa and an exponent can be formatted, how `xs:decimal` values with no digits after the decimal point are handled, and many, many other rules. See the spec for all the details.

- If the item is an `xs:date`, `xs:time`, or `xs:dateTime`, the string value is the default form of those datatypes. For example, an `xs:date` is represented as `YYYY-MM-DD-Z` in most cases (the `-Z` represents the timezone if one is defined), an `xs:time` is represented as `HH:MM:SS-Z` (where `Z` represents the timezone), and a `xs:dateTime` is represented as `YYYY-MM-DDTHH:MM:SS-Z`.

  The `string( )` function returns the raw form of these types; the `format-date( )`, `format-dateTime( )`, and `format-time( )` functions are more useful ways of converting these datatypes to strings.
- If the item is an `xs:dayTimeDuration`, `xs:yearMonthDuration`, or a basic `xs:duration`, the string value is the default format. An `xs:dayTimeDuration` is represented as `P2DT4H23M12.2S` (2 days, 4 hours, 23 minutes, and 12.2 seconds), for example. The functions `days-from-duration( )`, `hours-from-duration( )`, `minutes-from-duration( )`, `months-from-duration( )`, `seconds-from-duration( )`, and `years-from-duration( )` are much more useful ways of converting these datatypes to strings.
- If the item is an `xs:anyURI`, it is returned as its string value. None of the characters in the string value are escaped.
- For any other XML Schema datatypes that have a default format (typically known as the canonical representation), they are converted to `xs:string` values using that default format. The process for converting any datatypes that do not have a canonical representation to strings is implementation-defined.

Defined in

[1.0] XPath section 4.2, "String Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 2, "Accessors," with the rules for converting datatypes to strings defined in section 17.1.2, "Casting to `xs:string` and `xs:untypedAtomic`."

Example

We'll use our document of chocolate bar sales to test the `string( )` function:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
```

```
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

We'll test the `string( )` function with a variety of arguments:

```
<?xml version="1.0"?>
<!-- string.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the string() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  string(count(/report/brand)) = </xsl:text>
    <xsl:value-of select="string(count(/report/brand))"/>

    <xsl:text>&#xA;&#xA;  string(true()) = </xsl:text>
    <xsl:value-of select="string(true())"/>

    <xsl:text>&#xA;&#xA;  string(count(/report</xsl:text>
    <xsl:text>/brand/units[. &gt; 20000])) = </xsl:text>
    <xsl:value-of
      select="string(count(/report/brand/units[. &gt; 20000]))"/>

    <xsl:text>&#xA;&#xA;  string(/report/brand/units</xsl:text>
    <xsl:text>[. &gt; 20000][1]/../name) = </xsl:text>
    <xsl:value-of
      select="string(/report/brand/units[. &gt; 20000][1]/../name)"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results of our stylesheet:

```
Tests of the string() function:

  string(count(/report/brand)) = 5

  string(true()) = true

  string(count(/report/brand/units[. > 20000])) = 2

  string(/report/brand/units[. > 20000][1]/../name) = Lindt
```

[2.0] Converting a value to a string is usually pretty straightforward. Here is another stylesheet that converts some of XSLT 2.0's datatypes to strings:

```
<?xml version="1.0"?>
<!-- string2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;More tests of the string() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  string(xs:double(2718.28E-3)) = </xsl:text>
    <xsl:value-of select="string(xs:double(2718.28E-3))"/>

    <xsl:text>&#xA;&#xA;  string(current-time()) = </xsl:text>
    <xsl:value-of select="string(current-time())"/>

    <xsl:text>&#xA;&#xA;  string(xs:yearMonthDuration</xsl:text>
    <xsl:text>('P3Y8M')) = </xsl:text>
    <xsl:value-of
      select="string(xs:yearMonthDuration('P3Y8M'))"/>

    <xsl:text>&#xA;&#xA;  string(xs:date('1995-04-21')) = </xsl:text>
    <xsl:value-of
      select="string(xs:date('1995-04-21'))"/>

    <xsl:variable name="topBrands" as="xs:string*"
      select="/report/brand/units[. &gt; 15000]/../name/string()"/>
    <xsl:text>&#xA;&#xA;  Top-selling brands: &#xA;    </xsl:text>
    <xsl:value-of select="$topBrands" separator=", "/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
More tests of the string() function:

  string(xs:double(2718.28E-3)) = 2.71828

  string(current-time()) = 06:38:08.438-04:00

  string(xs:yearMonthDuration('P3Y8M')) = P3Y8M

  string(xs:date('1995-04-21')) = 1995-04-21

  Top-selling brands:
    Lindt, Valrhona, Ghirardelli
```

The stylesheet also uses the `string( )` function as the last step in an XPath expression. The expression `/report/brand/units[. > 15000]/../name/string( )` finds all of the `<units>` elements with a value greater than 15000, and then converts the text of their sibling `<name>` elements into strings, returning a sequence of strings. Outputting this sequence with the `<xsl:value-of>` element lists the three brands (Lindt, Valrhona, and Ghirardelli).

[2.0] string-join()

Given a sequence of strings and a separator string, returns a string containing each string from the sequence, separated by the second string.

Syntax

```
xs:string string-join(xs:string*, xs:string)
```

Inputs

A sequence of `xs:string`s and a string used as a separator.

Outputs

An `xs:string` that contains all of the strings in the sequence, separated by the second argument. If the second string is a zero-length string, the strings in the sequence are concatenated without a separator. If the sequence is the empty sequence, `string-join( )` returns a zero-length string.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 7.4, Functions on String Values.

Example

Here is a stylesheet that tests the `string-join( )` function:

```
<?xml version="1.0"?>
<!-- string-join.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the string-join() function:&#xA;</xsl:text>

    <xsl:variable name="testSequence" as="xs:string*">
      <xsl:sequence
        select="('Brooklyn', 'Manhattan', 'Williamsburg',
                 'George Washington', 'Tribeca')"/>
    </xsl:variable>

    <xsl:text>&#xA;  Our sequence of strings:&#xA;    </xsl:text>
    <xsl:value-of select="$testSequence" separator="&#xA;    "/>

    <xsl:text>&#xA;&#xA;  string-join($testSequeqnce, ' - ') = </xsl:text>
    <xsl:text>&#xA;    </xsl:text>
    <xsl:value-of select="string-join($testSequence, ' - ')"/>

    <xsl:text>&#xA;&#xA;  Courtesy of the specs, here's </xsl:text>
    <xsl:text>another example:&#xA;&#xA;    Ancestors of all </xsl:text>
    <xsl:text>the elements:&#xA;      using string-join&#xA;</xsl:text>
```

```
    <xsl:text>            (for $i in unordered</xsl:text>
    <xsl:text>(ancestor-or-self::*) return name($i), </xsl:text>
    <xsl:text>&#xA;             '/'):&#xA;</xsl:text>

    <xsl:for-each select="//*">
      <xsl:text>&#xA;      </xsl:text>
      <xsl:value-of
        select="string-join(for $i in ancestor-or-self::*
                return name($i), '/')"/>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
A test of the string-join() function:

  Our sequence of strings:
    Brooklyn
    Manhattan
    Williamsburg
    George Washington
    Tribeca

  string-join($testSequeqnce, ' - ') =
    Brooklyn - Manhattan - Williamsburg - George Washington - Tribeca

  Courtesy of the specs, here's another example:

    Ancestors of all the elements:
      using string-join
            (for $i in unordered(ancestor-or-self::*) return name($i),
             '/'):

      report
      report/title
      report/brand
      report/brand/name
      report/brand/units
      report/brand
      report/brand/name
      report/brand/units
      report/brand
      report/brand/name
      report/brand/units
      report/brand
      report/brand/name
      report/brand/units
      report/brand
      report/brand/name
      report/brand/units
```

string-length()

Returns the number of characters in the string passed in as the argument to this function. If no argument is specified, the context node is converted to a string and the length of that string is returned.

Syntax

```
[1.0] number string-length(string?)
[2.0] xs:integer string-length()
[2.0] xs:integer string-length(string?)
```

Inputs

An optional string.

Output

The number of characters defined in the string (an `xs:integer` in XSLT 2.0). If no string is specified as an argument, the context node is used. In other words, `string-length( )` and `string-length(.)` are equivalent.

[2.0] If the string is the empty sequence, the value `0` is returned.

Defined in

[1.0] XPath section 4.2, "String Functions."

[2.0] XQuery 1.0 and XPath 2.0 section 7.4, "Functions on String Values."

Example

The following example demonstrates the results of invoking the `string-length( )` function against various argument types. Here's the XML document we'll use for our example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
```

```
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

We'll process this document with the following stylesheet:

```
<?xml version="1.0"?>
<!-- string-length.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the string-length() function:</xsl:text>

    <xsl:text>&#xA;&#xA;   string-length() = </xsl:text>
    <xsl:value-of select="string-length()"/>
    <xsl:text>&#xA;   string-length(/report) = </xsl:text>
    <xsl:value-of select="string-length(/report)"/>
    <xsl:text>&#xA;   string-length(/report/brand[1]) = </xsl:text>
    <xsl:value-of select="string-length(/report/brand[1])"/>
    <xsl:text>&#xA;   string-length(/report/brand[1]/name) = </xsl:text>
    <xsl:value-of select="string-length(/report/brand[1]/name)"/>

    <xsl:text>&#xA;</xsl:text>
    <xsl:for-each select="/report/brand">
      <xsl:text>&#xA;   The name of brand #</xsl:text>
      <xsl:value-of select="position()"/>
      <xsl:text>, </xsl:text>
      <xsl:value-of select="name"/>
      <xsl:text>, has </xsl:text>
      <xsl:value-of select="string-length(name)"/>
      <xsl:text> characters.</xsl:text>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results of our stylesheet:

```
Tests of the string-length() function:

   string-length() = 168
   string-length(/report) = 168
   string-length(/report/brand[1]) = 23
   string-length(/report/brand[1]/name) = 5

   The name of brand #1, Lindt, has 5 characters.
   The name of brand #2, Callebaut, has 9 characters.
   The name of brand #3, Valrhona, has 8 characters.
   The name of brand #4, Perugina, has 8 characters.
   The name of brand #5, Ghirardelli, has 11 characters.
```

When we invoked the `string-length( )` function without any arguments, the root node (in XSLT 2.0, the document node) was converted to a string, and then the length of that string

was returned. (The context node is the root node at this point.) The string length of the root element (<report>) has the same value because there is no text outside the <report> element. Remember, the string value of any node is the concatenation of all the text in that node and all of its descendants.

[2.0] string-to-codepoints()

Converts a string into a sequence of Unicode codepoints.

Syntax

```
xs:integer* string-to-codepoints(xs:string?)
```

Inputs

An xs:string.

Output

A sequence of xs:integers, each of which represents a Unicode codepoint. If the input string is of zero length or is the empty sequence, the empty sequence is returned.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 7.2, "Functions to Assemble and Disassemble Strings."

Example

Here's a short stylesheet that converts strings into sequences of integers:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- string-to-codepoints.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the string-to-codepoints() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>&#xA;  string-to-codepoints('Lily') = </xsl:text>
    <xsl:value-of
      select="string-to-codepoints('Lily')" separator=", " />

    <xsl:text>&#xA;  string-to-codepoints('Straße') = </xsl:text>
    <xsl:value-of
      select="string-to-codepoints('Straße')" separator=", "/>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet generates these results:

```
Tests of the string-to-codepoints() function:

  string-to-codepoints('Lily') = 76, 105, 108, 121
  string-to-codepoints('Straße') = 83, 116, 114, 97, 223, 101
```

[2.0] subsequence()

Returns a contiguous portion of a sequence.

Syntax

```
item()* subsequence($sourceSeq as item()*, $startingLoc as xs:double)
item()* subsequence($sourceSeq as item()*, $startingLoc as xs:double,
                    $length as xs:double)
```

Inputs

A sequence and a starting position. An optional third argument specifies the number of items to be returned. The second and third arguments are `xs:double`s because many numeric operations on untyped data return `xs:double`.

Some details about how the two numeric arguments work:

- If the starting position is larger than the number of items in the sequence, the empty sequence is returned.
- If the sequence is the empty sequence, the empty sequence is returned.

Outputs

A new sequence containing the specified items.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.1, "General Functions and Operators on Sequences."

Example

Here's a stylesheet that illustrates how `subsequence( )` works:

```
<?xml version="1.0"?>
<!-- subsequence.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:variable name="testSequence" as="item()*">
      <xsl:sequence select="(3, 4, 5)"/>
```

```
  <xsl:element name="currentDate">
    <xsl:value-of select="current-date()"/>
  </xsl:element>
  <xsl:element name="currentTime">
    <xsl:value-of select="current-time()"/>
  </xsl:element>
  <xsl:element name="integerTest">
    <xsl:value-of select="xs:integer(8)"/>
  </xsl:element>
  <xsl:sequence select="('blue', 'red')"/>
  <xsl:element name="floatTest1">
    <xsl:value-of select="xs:float(3.14)"/>
  </xsl:element>
  <xsl:element name="floatTest2">
    <xsl:value-of select="xs:float(42)"/>
  </xsl:element>
  <xsl:element name="dateTest">
    <xsl:value-of select="xs:date('1995-04-21')"/>
  </xsl:element>
</xsl:variable>

<xsl:variable name="emptySequence" as="item()*">
  <xsl:sequence select="()"/>
</xsl:variable>

<xsl:text>&#xA;Here is a test of the subsequence() </xsl:text>
<xsl:text>function:&#xA;</xsl:text>

<xsl:text>&#xA;  Our original sequence is:&#xA;&#xA;    </xsl:text>
<xsl:value-of select="$testSequence"
  separator="&#xA;    "/>

<xsl:text>&#xA;&#xA;  subsequence($testSequence, 8) = </xsl:text>
<xsl:text>&#xA;    </xsl:text>
<xsl:value-of select="subsequence($testSequence, 8)"
  separator="&#xA;    "/>

<xsl:text>&#xA;&#xA;  subsequence($testSequence, 3, 3) = </xsl:text>
<xsl:text>&#xA;    </xsl:text>
<xsl:value-of select="subsequence($testSequence, 3, 3)"
  separator="&#xA;    "/>

<xsl:text>&#xA;&#xA;  subsequence($testSequence, 1, 1) = </xsl:text>
<xsl:text>&#xA;    </xsl:text>
<xsl:value-of select="subsequence($testSequence, 1, 1)"
  separator="&#xA;    "/>

<xsl:text>&#xA;&#xA;  subsequence($testSequence, -1, 3) = </xsl:text>
<xsl:text>&#xA;    </xsl:text>
<xsl:value-of select="subsequence($testSequence, -1, 3)"
  separator="&#xA;    "/>

<xsl:text>&#xA;&#xA;  subsequence($testSequence, 14, 2) = </xsl:text>
<xsl:text>&#xA;    </xsl:text>
<xsl:value-of select="subsequence($testSequence, 14, 2)"
```

```
      separator="&#xA;    "/>

    <xsl:text>&#xA;&#xA;  subsequence($testSequence, 2.3, 1.7) = </xsl:text>
    <xsl:text>&#xA;    </xsl:text>
    <xsl:value-of select="subsequence($testSequence, 2.3, 1.7)"
      separator="&#xA;    "/>

    <xsl:text>&#xA;&#xA;  subsequence($emptySequence, 1) = </xsl:text>
    <xsl:text>&#xA;    </xsl:text>
    <xsl:value-of select="subsequence($emptySequence, 1)"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Here is a test of the subsequence() function:

  Our original sequence is:

    3
    4
    5
    2007-02-19-05:00
    11:19:24.734-05:00
    8
    blue
    red
    3.14
    42
    1995-04-21

  subsequence($testSequence, 8) =
    red
    3.14
    42
    1995-04-21

  subsequence($testSequence, 3, 3) =
    5
    2007-02-19-05:00
    11:19:24.734-05:00

  subsequence($testSequence, 1, 1) =
    3

  subsequence($testSequence, -1, 3) =
    3

  subsequence($testSequence, 14, 2) =


  subsequence($testSequence, 2.3, 1.7) =
    4
    5
```

```
subsequence($emptySequence, 1) =
```

In the first example, we start at the eighth item in the sequence and return all of the subsequent items. Next, we start at the third item and return items 3, 4, and 5. To return a single item, we use `1, 1` to get one item starting at the first item. Using a negative start position can actually return items, as we see in the example `-1, 3`. The three items we're requesting are items –1, 0, and 1. Item 1 is the only one that exists, so that's what `subsequence( )` returns. In the next example, the starting position of 14 returns the empty sequence because there are less than 14 items. Using decimals for the numeric arguments means they are rounded to the nearest whole number, so this is the same as `2, 2`. Finally, calling `subsequence( )` against the empty sequence returns the empty sequence.

substring()

Returns a portion of a given string.

Syntax

```
[1.0] string substring($sourceString as string, $startingLoc as number,
                       $length as number?)
[2.0] xs:string substring($sourceString as xs:string?,
                          $startingLoc as xs:double)
[2.0] xs:string substring($sourceString as xs:string?,
                          $startingLoc as xs:double,
                          $length as xs:double)
```

Inputs

The `substring( )` function takes a string and one or two numbers as arguments. The string is the string from which the substring will be extracted. The second argument is used as the starting position of the returned substring, and the optional third argument specifies how many characters are returned.

Output

With two arguments (a string and a starting position), the `substring( )` function returns all characters in the string whose position is greater than or equal to the starting position. *Be aware that the first character in a string is at position 1, not 0.*

With three arguments (a string, a starting position, and a length), the `substring( )` function returns all characters in the string whose position is greater than or equal to the starting position or whose position is less than the starting position *plus* the length.

Normally, the arguments to the `substring( )` function are integers, although they may be floating-point numbers or more complicated expressions. In the case of floating-point numbers, the number is converted to an integer by the `round( )` function.

Defined in

[1.0] XPath section 4.2, "String Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 7.4, "Functions on String Values."

Example

Here's the stylesheet we'll use to demonstrate the `substring( )` function:

```
<?xml version="1.0"?>
<!-- substring.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the substring() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  1. substring('Now is the time', 4)     = "</xsl:text>
    <xsl:value-of select="substring('Now is the time', 4)"/>
    <xsl:text>"&#xA;</xsl:text>
    <xsl:text>  2. substring('Now is the time', 4, 6)  = "</xsl:text>
    <xsl:value-of select="substring('Now is the time', 4, 6)"/>
    <xsl:text>"&#xA;</xsl:text>

    <!-- Xalan chokes on this example, Saxon and MSXSL don't. -->
    <xsl:text>  3. substring('Now is the time', 8, -6) = "</xsl:text>
    <xsl:value-of select="substring('Now is the time', 8, -6)"/>
    <xsl:text>"&#xA;</xsl:text>
    <xsl:text>  4. substring('Now is the time', -3, 6) = "</xsl:text>
    <xsl:value-of select="substring('Now is the time', -3, 6)"/>
    <xsl:text>"&#xA;</xsl:text>
    <xsl:text>  5. substring('Now is the time', 54, 6) = "</xsl:text>
    <xsl:value-of select="substring('Now is the time', 54, 6)"/>
    <xsl:text>"&#xA;&#xA;</xsl:text>
    <xsl:text>  6. substring('Now is the time', 1.7, 6.4) = "</xsl:text>
    <xsl:value-of select="substring('Now is the time', 1.7, 6.4)"/>
    <xsl:text>"&#xA;</xsl:text>
    <xsl:text>  7. substring('Now is the time', 1.7, 6.5) = "</xsl:text>
    <xsl:value-of select="substring('Now is the time', 1.7, 6.5)"/>
    <xsl:text>"&#xA;&#xA;</xsl:text>
    <xsl:text>  count(document('')//*) = </xsl:text>
    <xsl:value-of select="count(document('')//*)"/>
    <xsl:text>&#xA;&#xA;  8. substring('Here is a really, </xsl:text>
    <xsl:text>really, really long string',</xsl:text>
    <xsl:text>&#xA;     count(document('')//*)) = "</xsl:text>
    <xsl:value-of
      select="substring('Here is a really, really, really long string',
              count(document('')//*))"/>
    <xsl:text>"&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

When using the Saxon or MSXSL processor, here are the results:

```
Tests of the substring() function:

  1. substring('Now is the time', 4) = " is the time"
  2. substring('Now is the time', 4, 6) = " is th"
  3. substring('Now is the time', 8, -6) = ""
  4. substring('Now is the time', -3, 6) = "No"
  5. substring('Now is the time', 54, 6) = ""

  6. substring('Now is the time', 1.7, 6.4) = "ow is "
  7. substring('Now is the time', 1.7, 6.5) = "ow is t"

  count(document('')//*) = 32

  8. substring('Here is a really, really, really long string',
     count(document('')//*)) = "y long string"
```

We'll look at each test case individually:

1. The first test case returns all the characters whose position is `>= 4`.
2. The second test case returns all the characters whose position is `>= 4` and `< 10`. (Remember, the starting value and the length are added together to determine the upper bound of the substring.)
3. The third test case returns all the characters whose position is `>= 8` and `< 2`. This useless range of characters is, of course, empty.
4. This test returns all the characters whose position is `>= -3` and `< 3`. Even though starting from a negative position seems as though it would return nothing, positions 1 and 2 fall into the requested range.
5. This test returns nothing because the starting position is greater than the length of the string.
6. This test case uses floating-point numbers for the starting position and the length. The two numbers are rounded, meaning this is equivalent to `substring('Now is the time', 2, 6)`. All characters whose positions are `>= 2` and `< 8` are returned.
7. This test case also uses floating-point numbers. In this case, the second number (`6.5`) is rounded to 7, making this equivalent to `substring('Now is the time', 2, 7)`. Characters whose positions are `>= 2` and `< 9` are returned.
8. For the final two test cases, we use a more complicated XPath expression (`count(document('')//*)`) to generate a number. This value is the number of elements in the stylesheet itself. This test case returns all characters whose position is `>= 36`.

A final note: Xalan raises an error with the example `substring('Now is the time', 8, -6)`:

```
XSLT Error (javax.xml.transform.TransformerException):
java.lang.StringIndexOutOfBoundsException:
String index out of range: -6
```

Although this is an error in Xalan's implementation, it does make the point that passing unreasonable data to a function can lead to unpleasant results.

substring-after()

Given a data string and a search string, returns the portion of the data string after the first occurrence of the search string. If the search string is not found, the `substring-after( )` function returns a zero-length string.

Syntax

```
[1.0] string substring-after(string, string)
[2.0] xs:string substring-after(xs:string?, xs:string?)
[2.0] xs:string substring-after(xs:string?, xs:string?,
                                $collation as xs:string)
```

Inputs

Two strings. The first string is the data string to be searched, and the second is the search string. `substring-after( )` looks for the search string in the data string.

[2.0] In XSLT 2.0, there is an optional third argument: the name of a collation that specifies how strings are compared.

Output

The portion of the data string that occurs after the first occurrence of the search string. If the search string does not appear in the data string, the function returns an empty string. If the second string is a zero-length string, `substring-after( )` returns a zero-length string. The function also returns a zero-length string if the first string is shorter than the second string.

Defined in

[1.0] XPath section 4.2, "String Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 7.5, "Functions Based on Substring Matching."

Example

Here is a sample stylesheet that uses `substring-after( )`:

```
<?xml version="1.0"?>
<!-- substring-after1.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the substring-after() </xsl:text>
    <xsl:text>function:&#xA;&#xA;</xsl:text>

    <xsl:text>  substring-after('Abracadabra', </xsl:text>
```

```
    <xsl:text>'bra')  = "</xsl:text>
    <xsl:value-of
      select="substring-after('Abracadabra', 'bra')"/>
    <xsl:text>"&#xA;</xsl:text>

    <xsl:text>  substring-after('Abracadabra', </xsl:text>
    <xsl:text>'abra') = "</xsl:text>
    <xsl:value-of
      select="substring-after('Abracadabra', 'abra')"/>
    <xsl:text>"&#xA;</xsl:text>

    <xsl:text>  substring-after('Abracadabra', </xsl:text>
    <xsl:text>'A')    = "</xsl:text>
    <xsl:value-of
      select="substring-after('Abracadabra', 'A')"/>
    <xsl:text>"&#xA;</xsl:text>

    <xsl:text>  substring-after('Abracadabra', </xsl:text>
    <xsl:text>'A')    = "</xsl:text>
    <xsl:value-of
      select="substring-after('Abracadabra', 'a')"/>
    <xsl:text>"&#xA;</xsl:text>

    <xsl:text>  substring-after('Abracadabra', </xsl:text>
    <xsl:text>'')     = "</xsl:text>
    <xsl:value-of
      select="substring-after('Abracadabra', '')"/>
    <xsl:text>"&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet produces these results:

```
A test of the substring-after() function:

  substring-after('Abracadabra', 'bra')  = "cadabra"
  substring-after('Abracadabra', 'abra') = ""
  substring-after('Abracadabra', 'A')    = "bracadabra"
  substring-after('Abracadabra', 'A')    = "cadabra"
  substring-after('Abracadabra', '')     = "Abracadabra"
```

As a second example, here's an XSLT 2.0 stylesheet that uses a custom collation:

```
<?xml version="1.0"?>
<!-- substring-after2.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Another test of the substring-after() </xsl:text>
    <xsl:text>function:&#xA;&#xA;</xsl:text>

    <xsl:text>  substring-after('Schoenaicherstrasse', </xsl:text>
    <xsl:text>'Sch&#xF6;n') = &#xA;    "</xsl:text>
```

```
    <xsl:value-of
      select="substring-after('Schoenaicherstrasse', 'Sch&#xF6;n')"/>
    <xsl:text>"&#xA;</xsl:text>

    <xsl:text>  substring-after('Schoenaicherstrasse', </xsl:text>
    <xsl:text>'Sch&#xF6;n', [German collation]) = &#xA;    "</xsl:text>
    <xsl:value-of
      select="substring-after('Schoenaicherstrasse', 'Sch&#xF6;n',
              concat('http://saxon.sf.net/collation?',
                     'class=com.oreilly.xslt.GermanCollation;'))"/>
    <xsl:text>"&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet generates these results:

```
Another test of the substring-after() function:

  substring-after('Schoenaicherstrasse', 'Schön') =
    ""
  substring-after('Schoenaicherstrasse', 'Schön', [German collation]) =
    "aicherstrasse"
```

The German umlaut-o character (*ö*) is equivalent to *oe*. The default collation doesn't recognize this, but our custom collation does. (To keep the listing within the margins of the page, we used the `concat( )` function to combine the two halves of the Saxon collation URI.) See "The document() Function and Sorting" in Chapter 8 for more information.

substring-before()

Given a data string and a search string, returns the portion of the data string before the first occurrence of the search string. If the search string is not found, the `substring-before( )` function returns a zero-length string.

Syntax

```
[1.0] string substring-before(string, string)
[2.0] xs:string substring-before(xs:string?, xs:string?)
[2.0] xs:string substring-before(xs:string?, xs:string?,
                                 $collation as xs:string)
```

Inputs

Two strings. The first string is the string to be searched, and the second is the string to be searched for in the first string.

[2.0] In XSLT 2.0, there is an optional third argument: the name of a collation that specifies how strings are compared.

Output

The portion of the data string that occurs before the first occurrence of the search string. If the search string does not appear in the data string, the function returns an empty string. If

the second string is a zero-length string, `substring-before( )` returns a zero-length string. The function also returns a zero-length string if the first string is shorter than the second string.

Defined in

[1.0] XPath section 4.2, "String Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 7.5, "Functions Based on Substring Matching."

Example

Here is a sample stylesheet that uses `substring-before( )`:

```
<?xml version="1.0"?>
<!-- substring-before1.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the substring-before() </xsl:text>
    <xsl:text>function:&#xA;&#xA;</xsl:text>

    <xsl:text>  substring-before('Abracadabra', </xsl:text>
    <xsl:text>'abra')        = "</xsl:text>
    <xsl:value-of
      select="substring-before('Abracadabra', 'abra')"/>
    <xsl:text>"&#xA;</xsl:text>

    <xsl:text>  substring-before('Abracadabra', </xsl:text>
    <xsl:text>'Abracadabra') = "</xsl:text>
    <xsl:value-of
      select="substring-before('Abracadabra', 'Abracadabra')"/>
    <xsl:text>"&#xA;</xsl:text>

    <xsl:text>  substring-before('Abracadabra', </xsl:text>
    <xsl:text>'A')           = "</xsl:text>
    <xsl:value-of
      select="substring-before('Abracadabra', 'A')"/>
    <xsl:text>"&#xA;</xsl:text>

    <xsl:text>  substring-before('Abracadabra', </xsl:text>
    <xsl:text>'a')           = "</xsl:text>
    <xsl:value-of
      select="substring-before('Abracadabra', 'a')"/>
    <xsl:text>"&#xA;</xsl:text>

    <xsl:text>  substring-before('Abracadabra', </xsl:text>
    <xsl:text>'')            = "</xsl:text>
    <xsl:value-of
      select="substring-before('Abracadabra', '')"/>
    <xsl:text>"&#xA;</xsl:text>
  </xsl:template>
```

```
</xsl:stylesheet>
```

The stylesheet produces these results:

```
A test of the substring-before() function:

  substring-before('Abracadabra', 'abra')        = "Abracad"
  substring-before('Abracadabra', 'Abracadabra') = ""
  substring-before('Abracadabra', 'A')           = ""
  substring-before('Abracadabra', 'a')           = "Abr"
  substring-before('Abracadabra', '')            = ""
```

As a second example, here's an XSLT 2.0 stylesheet that uses a custom collation:

```
<?xml version="1.0"?>
<!-- substring-before2.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Another test of the substring-before() </xsl:text>
    <xsl:text>function:&#xA;&#xA;</xsl:text>

    <xsl:text>  substring-before('Schoenaicherstrasse', </xsl:text>
    <xsl:text>'stra&#xDF;e') = &#xA;    "</xsl:text>
    <xsl:value-of
      select="substring-before('Schoenaicherstrasse', 'stra&#xDF;e')"/>
    <xsl:text>"&#xA;</xsl:text>

    <xsl:text>  substring-before('Schoenaicherstrasse', </xsl:text>
    <xsl:text>'stra&#xDF;e', [German collation]) = &#xA;    "</xsl:text>
    <xsl:value-of
      select="substring-before('Schoenaicherstrasse', 'stra&#xDF;e',
              concat('http://saxon.sf.net/collation?',
                     'class=com.oreilly.xslt.GermanCollation;'))"/>
    <xsl:text>"&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet generates these results:

```
Another test of the substring-before() function:

  substring-before('Schoenaicherstrasse', 'straße') =
    ""
  substring-before('Schoenaicherstrasse', 'straße', [German collation]) =
    "Schoenaicher"
```

The German sharp-s character (*ß*) is equivalent to *ss*. The default collation doesn't recognize this, but our custom collation does. (To keep the listing within the margins of the page, we used the `concat( )` function to combine the two halves of the Saxon collation URI.) See "The document() Function and Sorting" in Chapter 8 for more information.

sum()

Converts all nodes in the argument node-set to numbers, and then returns the sum of all of those numbers.

Syntax

```
[1.0] number sum(node-set)
[2.0] xs:anyAtomicType sum(xs:anyAtomicType)
[2.0] xs:anyAtomicType sum(xs:anyAtomicType, $empty as xs:anyAtomicType)
```

Inputs

[1.0] A node-set. Any node in the node-set that is not a number is converted to a number as if it were passed to the `number( )` function, after which the numeric values of all of the nodes are summed.

[2.0] A sequence of atomic values, plus an optional value that is returned when the first argument to `sum( )` is the empty sequence.

Output

[1.0] The sum of the numeric values of all of the nodes in the argument node-set. If any node in the argument node-set cannot be converted to a number, the `sum( )` function returns `NaN`.

[2.0] Given a sequence of numeric values, `sum( )` returns the sum of those values. Given a sequence of durations, `sum( )` returns the sum of those durations. As you'd expect from XSLT 2.0, there are some complications to consider:

- To calculate the sum of a sequence of durations, the durations must all be `xs:dayTimeDuration`s, or they must all be `xs:yearMonthDuration`s. You can't mix the two types of durations; if you do, the XSLT processor raises an error.
- If any of the items in the sequence are of type `xs:untypedAtomic`, the XSLT processor attempts to cast it to `xs:double`. If it can't be converted to an `xs:double`, the XSLT processor raises an error.
- If you pass a nonempty sequence to `sum( )` and the sum is `0`, the function returns `0`.
- Passing the empty sequence to `sum( )` returns `0` by default. If for some reason you'd like the empty sequence to return some other value, such as the string `"Not applicable"`, you can specify the optional `$empty` argument. That argument, if it exists, is returned when you call `sum( )` with the empty sequence.

Defined in

[1.0] XPath section 4.4, "Number Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 15.4, "Aggregate Functions."

Example

We'll demonstrate the `sum( )` function against the following XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

Here is a very simple stylesheet that uses the `sum( )` function:

```
<?xml version="1.0"?>
<!-- sum.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the sum() function:&#xA;&#xA;</xsl:text>

    <xsl:text>  Total chocolate bar sales this quarter: </xsl:text>
    <xsl:value-of
      select="format-number(sum(/report/brand/units), '###,###')"/>
  </xsl:template>

</xsl:stylesheet>
```

Processing the XML document with this stylesheet generates these results:

```
A test of the sum() function:

  Total chocolate bar sales this quarter: 91,316
```

We used the `format-number( )` function to format the value returned by the `sum( )` function.

Here's an XSLT 2.0 stylesheet that uses the `sum( )` function against a variety of sequences:

```
<?xml version="1.0"?>
<!-- sum2.xsl -->
<xsl:stylesheet version="2.0"
```

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xsl:output method="text"/>

<xsl:template match="/">
  <xsl:variable name="seq1" select="(3, 5, 18)"/>
  <xsl:variable name="seq2" select="(3, 5, 48.273, 2.9e3)"/>

  <xsl:variable name="value1" as="xs:integer" select="42"/>
  <xsl:variable name="value2" as="xs:double" select="2718.28E-3"/>
  <xsl:variable name="value3" as="xs:float" select="98.6"/>
  <xsl:variable name="value4" as="xs:decimal" select="2.54"/>
  <xsl:variable name="seq3"
    select="($value1, $value2, $value3, $value4)"/>

  <xsl:variable name="seq4"
    select="(xs:yearMonthDuration('P3Y8M'),
             xs:yearMonthDuration('P4Y2M'),
             xs:yearMonthDuration('P6Y4M'))"/>
  <xsl:variable name="seq5"
    select="(xs:dayTimeDuration('P2DT4H23M12.2S'),
             xs:dayTimeDuration('P3DT8H17M'),
             xs:dayTimeDuration('P3D'))"/>

  <xsl:text>Here are some tests of the sum() function:&#xA;</xsl:text>

  <xsl:text>&#xA;  sum(</xsl:text>
  <xsl:value-of select="$seq1" separator=", "/>
  <xsl:text>) = </xsl:text>
  <xsl:value-of select="format-number(sum($seq1), '#.###')"/>

  <xsl:text>&#xA;&#xA;  sum(</xsl:text>
  <xsl:value-of select="$seq2" separator=", "/>
  <xsl:text>) = </xsl:text>
  <xsl:value-of select="format-number(sum($seq2), '#.###')"/>

  <xsl:text>&#xA;&#xA;  sum(</xsl:text>
  <xsl:value-of select="$seq3" separator=", "/>
  <xsl:text>) = </xsl:text>
  <xsl:value-of select="format-number(sum($seq3), '#.###')"/>

  <xsl:text>&#xA;&#xA;  sum(</xsl:text>
  <xsl:value-of select="$seq4" separator=", "/>
  <xsl:text>) = </xsl:text>
  <xsl:value-of select="sum($seq4)"/>

  <xsl:text>&#xA;&#xA;    In text, the sum of</xsl:text>
  <xsl:for-each select="$seq4">
    <xsl:text>&#xA;      </xsl:text>
    <xsl:value-of select="years-from-duration(.)"/>
    <xsl:text> years and </xsl:text>
    <xsl:value-of select="months-from-duration(.)"/>
    <xsl:text> months (</xsl:text>
    <xsl:value-of select="."/>
```

```
    <xsl:text>)</xsl:text>
  </xsl:for-each>

  <xsl:text>&#xA;    is </xsl:text>
  <xsl:value-of select="years-from-duration(sum($seq4))"/>
  <xsl:text> years and </xsl:text>
  <xsl:value-of select="months-from-duration(sum($seq4))"/>
  <xsl:text> months (</xsl:text>
  <xsl:value-of select="sum($seq4)"/>
  <xsl:text>).</xsl:text>

  <xsl:text>&#xA;&#xA;  sum(</xsl:text>
  <xsl:value-of select="$seq5" separator=", "/>
  <xsl:text>) = </xsl:text>
  <xsl:variable name="sum5" select="sum($seq5)"/>
  <xsl:value-of select="$sum5"/>

  <xsl:text>&#xA;&#xA;    In text, the sum of</xsl:text>
  <xsl:for-each select="$seq5">
    <xsl:text>&#xA;      </xsl:text>
    <xsl:value-of select="days-from-duration(.)"/>
    <xsl:text> days, </xsl:text>
    <xsl:value-of select="hours-from-duration(.)"/>
    <xsl:text> hours, </xsl:text>
    <xsl:value-of select="minutes-from-duration(.)"/>
    <xsl:text> minutes and </xsl:text>
    <xsl:value-of
      select="format-number(seconds-from-duration(.), '#.##')"/>
    <xsl:text> seconds (</xsl:text>
    <xsl:value-of select="."/>
    <xsl:text>)</xsl:text>
  </xsl:for-each>

  <xsl:text>&#xA;    is </xsl:text>
  <xsl:value-of select="days-from-duration($sum5)"/>
  <xsl:text> days, </xsl:text>
  <xsl:value-of select="hours-from-duration($sum5)"/>
  <xsl:text> hours, </xsl:text>
  <xsl:value-of select="minutes-from-duration($sum5)"/>
  <xsl:text> minutes and </xsl:text>
  <xsl:value-of
    select="format-number(seconds-from-duration($sum5), '#.##')"/>
  <xsl:text> seconds.</xsl:text>

  <xsl:text>&#xA;&#xA;  sum(()) = </xsl:text>
  <xsl:value-of select="sum(())"/>

  <xsl:text>&#xA;&#xA;  sum((), 'Not applicable') = </xsl:text>
  <xsl:value-of select="sum((), 'Not applicable')"/>

  <xsl:text>&#xA;&#xA;  sum((3, -3), 'Not applicable') = </xsl:text>
  <xsl:value-of select="sum((3, -3), 'Not applicable')"/>

</xsl:template>
```

```
</xsl:stylesheet>
```

Here are the results from the XSLT 2.0 stylesheet:

```
Here are some tests of the sum() function:

  sum(3, 5, 18) = 26

  sum(3, 5, 48.273, 2900) = 2956.273

  sum(42, 2.71828, 98.6, 2.54) = 145.858

  sum(P3Y8M, P4Y2M, P6Y4M) = P14Y2M

    In text, the sum of
      3 years and 8 months (P3Y8M)
      4 years and 2 months (P4Y2M)
      6 years and 4 months (P6Y4M)
    is 14 years and 2 months (P14Y2M).

  sum(P2DT4H23M12.2S, P3DT8H17M, P3D) = P8DT12H40M12.2S

    In text, the sum of
      2 days, 4 hours, 23 minutes and 12.2 seconds (P2DT4H23M12.2S)
      3 days, 8 hours, 17 minutes and 0 seconds (P3DT8H17M)
      3 days, 0 hours, 0 minutes and 0 seconds (P3D)
    is 8 days, 12 hours, 40 minutes and 12.2 seconds.

  sum(()) = 0

  sum((), 'Not applicable') = Not applicable

  sum((3, -3), 'Not applicable') = 0
```

system-property()

Returns the value of the system property named by the argument to the function.

Syntax

```
[1.0] object system-property(string)
[2.0] xs:string system-property(xs:string)
```

Inputs

The three properties defined by the XSLT 1.0 specification must be supported by all XSLT 1.0 processors; XSLT 2.0 processors must support the five additional properties defined in the XSLT 2.0 spec. Other properties may be supported by individual processors, although vendors are not allowed to add more properties in the `xsl:` namespace.

All XSLT processors must support three system properties:

`xsl:version`
: A floating-point number representing the version of XSLT implemented by this XSLT processor. This property is equivalent to the numeric value `1.0`, although some XSLT processors (Xalan, for example) return the value `1`. XSLT 2.0 processors must report the version as a floating-point number such as `2.0`.

`xsl:vendor`
: A string identifying the vendor of this XSLT processor.

`xsl:vendor-url`
: A string containing the URL identifying the vendor of the XSLT processor. This string is typically the home page of the vendor's web site.

[2.0] The XSLT 2.0 specification defines five additional properties that must be supported by all XSLT 2.0 processors:

`xsl:product-name`
: The vendor's name for the processor. The XSLT 2.0 specification recommends that this name remain constant from one release of the product to the next, although this is not a requirement.

`xsl:product-version`
: The vendor-defined version of the processor. The XSLT 2.0 spec does not require that the version be in any particular format.

`xsl:is-schema-aware`
: Returns the string `yes` if the XSLT processor is schema-aware; returns the string `no` otherwise.

`xsl:supports-serialization`
: Returns the string `yes` if the XSLT processor supports serialization; returns the string `no` otherwise.

`xsl:supports-backwards-compatibility`
: Returns the string `yes` if the XSLT processor supports backwards compatibility for earlier versions of XSLT; returns the string `no` otherwise.

Output

The value of the queried property.

Defined in

[1.0] XSLT section 12.4, "Miscellaneous Additional Functions."

[2.0] XSLT section 16.6, "Miscellaneous Additional Functions."

Example

Here is a stylesheet that queries the properties defined by XSLT 1.0:

```
<?xml version="1.0"?>
<!-- system-property1.xsl -->
<xsl:stylesheet version="1.0"
```

```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="text"/>

    <xsl:template match="/">
      <xsl:text>&#xA;A test of the system-property() function:</xsl:text>
      <xsl:text>&#xA;&#xA;  xsl:version = "</xsl:text>
      <xsl:value-of select="system-property('xsl:version')"/>
      <xsl:text>"&#xA;</xsl:text>
      <xsl:text>  xsl:vendor = "</xsl:text>
      <xsl:value-of select="system-property('xsl:vendor')"/>
      <xsl:text>"&#xA;</xsl:text>
      <xsl:text>  xsl:vendor-url = "</xsl:text>
      <xsl:value-of select="system-property('xsl:vendor-url')"/>
      <xsl:text>"</xsl:text>
    </xsl:template>

  </xsl:stylesheet>
```

We'll process this stylesheet with the Saxon processor:

```
A test of the system-property() function:

  xsl:version = "2.0"
  xsl:vendor = "SAXON 9.0.0.3 from Saxonica"
  xsl:vendor-url = "http://www.saxonica.com/"
```

Running this stylesheet with Xalan-J gives these results:

```
A test of the system-property() function:

  xsl:version = "1"
  xsl:vendor = "Apache Software Foundation (Xalan XSLTC)"
  xsl:vendor-url = "http://xml.apache.org/xalan-j"
```

The Microsoft XSLT processor generates these values:

```
A test of the system-property() function:

  xsl:version = "1"
  xsl:vendor = "Microsoft"
  xsl:vendor-url = "http://www.microsoft.com"
```

Finally, we'll use the Altova XML engine in XSLT 1.0 mode:

```
A test of the system-property() function:

  xsl:version = "1"
  xsl:vendor = "Altova GmbH"
  xsl:vendor-url = "http://www.altova.com"
```

In XSLT 2.0 mode, the Altova processor returns `xsl:version = "2.0"`.

Here's a stylesheet that tests for the XSLT 2.0 properties as well:

```
<?xml version="1.0"?>
<!-- system-property2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the system-property() function:</xsl:text>
    <xsl:text>&#xA;&#xA;  xsl:version = "</xsl:text>
    <xsl:value-of select="system-property('xsl:version')"/>
    <xsl:text>"&#xA;</xsl:text>
    <xsl:text>  xsl:vendor = "</xsl:text>
    <xsl:value-of select="system-property('xsl:vendor')"/>
    <xsl:text>"&#xA;</xsl:text>
    <xsl:text>  xsl:vendor-url = "</xsl:text>
    <xsl:value-of select="system-property('xsl:vendor-url')"/>
    <xsl:text>"&#xA;&#xA;XSLT 2.0 properties:&#xA;&#xA;</xsl:text>
    <xsl:text>  xsl:product-name = "</xsl:text>
    <xsl:value-of select="system-property('xsl:product-name')"/>
    <xsl:text>"&#xA;</xsl:text>
    <xsl:text>  xsl:product-version = "</xsl:text>
    <xsl:value-of select="system-property('xsl:product-version')"/>
    <xsl:text>"&#xA;</xsl:text>
    <xsl:text>  xsl:is-schema-aware = "</xsl:text>
    <xsl:value-of select="system-property('xsl:is-schema-aware')"/>
    <xsl:text>"&#xA;</xsl:text>
    <xsl:text>  xsl:supports-serialization = "</xsl:text>
    <xsl:value-of
      select="system-property('xsl:supports-serialization')"/>
    <xsl:text>"&#xA;</xsl:text>
    <xsl:text>  xsl:supports-backwards-compatibility = "</xsl:text>
    <xsl:value-of
      select="system-property('xsl:supports-backwards-compatibility')"/>
    <xsl:text>"</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

When processing this stylesheet with Saxon-B, the non-schema-aware processor, the results look like this:

```
A test of the system-property() function:

  xsl:version = "2.0"
  xsl:vendor = "SAXON 9.0.0.3 from Saxonica"
  xsl:vendor-url = "http://www.saxonica.com/"

XSLT 2.0 properties:

  xsl:product-name = "SAXON"
  xsl:product-version = "9.0.0.3"
  xsl:is-schema-aware = "no"
  xsl:supports-serialization = "yes"
  xsl:supports-backwards-compatibility = "yes"
```

The same stylesheet with the schema-aware version (Saxon-SA) generates these results:

```
A test of the system-property() function:

  xsl:version = "2.0"
```

```
  xsl:vendor = "SAXON 9.0.0.3 from Saxonica"
  xsl:vendor-url = "http://www.saxonica.com/"

XSLT 2.0 properties:

  xsl:product-name = "SAXON"
  xsl:product-version = "SA 9.0.0.3"
  xsl:is-schema-aware = "yes"
  xsl:supports-serialization = "yes"
  xsl:supports-backwards-compatibility = "yes"
```

Processing the stylesheet with the Altova XML engine in XSLT 2.0 mode gives these results:

```
A test of the system-property() function:

  xsl:version = "2.0"
  xsl:vendor = "Altova GmbH"
  xsl:vendor-url = "http://www.altova.com"

XSLT 2.0 properties:

  xsl:product-name = "Altova XSLT Engine"
  xsl:product-version = "2007"
  xsl:is-schema-aware = "yes"
  xsl:supports-serialization = "yes"
  xsl:supports-backwards-compatibility = "no"
```

Finally, XSLT processors are free to support other properties. The Java version of Saxon, for example, supports the Java system properties. Here's a stylesheet that queries some of those properties:

```
<?xml version="1.0"?>
<!-- system-property3.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Getting Java system properties with </xsl:text>
    <xsl:text>system-property():</xsl:text>
    <xsl:text>&#xA;&#xA;  java.version = "</xsl:text>
    <xsl:value-of select="system-property('java.version')"/>
    <xsl:text>"&#xA;  path.separator = "</xsl:text>
    <xsl:value-of select="system-property('path.separator')"/>
    <xsl:text>"&#xA;  file.separator = "</xsl:text>
    <xsl:value-of select="system-property('file.separator')"/>
    <xsl:text>"&#xA;  user.name = "</xsl:text>
    <xsl:value-of select="system-property('user.name')"/>
    <xsl:text>"&#xA;  user.country = "</xsl:text>
    <xsl:value-of select="system-property('user.country')"/>
    <xsl:text>"</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results Saxon returns:

```
Getting Java system properties with system-property():

  java.version = "1.5.0_10"
  path.separator = ";"
  file.separator = "\"
  user.name = "Skippy"
  user.country = "US"
```

Using this stylesheet with Altova returns blank values for all of the Java properties.

[2.0] timezone-from-date()

Given an `xs:date` value, returns its timezone component.

Syntax

```
xs:dayTimeDuration? timezone-from-date(xs:date?)
```

Input

An `xs:date` value.

Output

An `xs:dayTimeDuration` that represents the timezone component of the given `xs:date` value. If the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet illustrates the `timezone-from-date( )` function:

```
<?xml version="1.0"?>
<!-- timezone-from-date.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the timezone from an xs:date:</xsl:text>
    <xsl:variable name="currentDate" as="xs:date" select="current-date()"/>
    <xsl:text>&#xA;&#xA;  The current date is: </xsl:text>
    <xsl:value-of select="$currentDate"/>

    <xsl:text>&#xA;&#xA;  The current timezone is: </xsl:text>
    <xsl:value-of select="timezone-from-date($currentDate)"/>
    <xsl:text>&#xA;    The timezone is also known as </xsl:text>
    <xsl:value-of select="format-date($currentDate, '[ZN]')"/>
  </xsl:template>
```

```
</xsl:stylesheet>
```

The stylesheet creates these results:

```
Extracting the timezone from an xs:date:

  The current date is: 2006-11-16-05:00

  The current timezone is: -PT5H
    The timezone is also known as EST
```

Notice that the first result was generated by the `timezone-from-date( )` function, while the second was generated using `format-date( )` with a format string that selected only the timezone component of the `xs:date` value.

See Also

The definitions of the [2.0] `day-from-date( )`, [2.0] `format-date( )`, [2.0] `month-from-date( )`, and [2.0] `year-from-date( )` functions.

[2.0] timezone-from-dateTime()

Given an `xs:dateTime` value, returns its timezone.

Syntax

```
xs:dayTimeDuration? timezone-from-dateTime(xs:dateTime?)
```

Inputs

An `xs:dateTime` value.

Output

An `xs:dayTimeDuration` representing the timezone of the given `xs:dateTime` value. If the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `timezone-from-dateTime( )` function:

```
<?xml version="1.0"?>
<!-- timezone-from-datetime.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>
```

```
  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the timezone from an xs:dateTime:</xsl:text>
    <xsl:variable name="currentDateTime" as="xs:dateTime"
      select="current-dateTime()"/>
    <xsl:text>&#xA;&#xA;  The current date and time is: </xsl:text>
    <xsl:value-of select="$currentDateTime"/>

    <xsl:text>&#xA;&#xA;  The current timezone is: </xsl:text>
    <xsl:value-of select="timezone-from-dateTime($currentDateTime)"/>
    <xsl:text>&#xA;    The timezone is also known as </xsl:text>
    <xsl:value-of select="format-dateTime($currentDateTime, '[ZN]')"/>

  </xsl:template>

</xsl:stylesheet>
```

The stylesheet creates these results:

```
Extracting the timezone from an xs:dateTime:

  The current date and time is: 2006-11-16T05:13:44.431-05:00

  The current timezone is: -PT5H
    The timezone is also known as EST
```

Notice that the first result was generated by the `timezone-from-dateTime( )` function, while the second was generated by using `format-dateTime( )` with a format string that selected only the timezone component of the `xs:dateTime` value.

See Also

The definitions of the [2.0] `day-from-dateTime( )`, [2.0] `format-dateTime( )`, [2.0] `hours-from-dateTime( )`, [2.0] `minutes-from-dateTime( )`, [2.0] `month-from-dateTime( )`, [2.0] `seconds-from-dateTime( )`, and [2.0] `year-from-dateTime( )` functions.

[2.0] timezone-from-time()

Given an `xs:time` value, returns its timezone component.

Syntax

```
xs:dayTimeDuration? timezone-from-time(xs:time?)
```

Input

An `xs:time` value.

Output

An `xs:dayTimeDuration` value representing the timezone component of the given `xs:time` value. If the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet illustrates the `timezone-from-time( )` function:

```
<?xml version="1.0"?>
<!-- timezone-from-time.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the timezone from an xs:time:</xsl:text>
    <xsl:variable name="currentTime" as="xs:time" select="current-time()"/>
    <xsl:text>&#xA;&#xA;  The current time is: </xsl:text>
    <xsl:value-of select="$currentTime"/>

    <xsl:text>&#xA;&#xA;  The current timezone is: </xsl:text>
    <xsl:value-of select="timezone-from-time($currentTime)"/>
    <xsl:text>&#xA;    The timezone is also known as </xsl:text>
    <xsl:value-of select="format-time($currentTime, '[ZN]')"/>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet creates these results:

```
Extracting the timezone component from an xs:time:

  The current time is: 05:15:30.423-05:00

  The current timezone is: -PT5H
    The timezone is also known as EST
```

Notice that the first result was generated by `timezone-from-time( )`, while the second was generated with the `format-time( )` function and a format string that selected the timezone component of the `xs:time`.

See Also

The definitions of the [2.0] `hours-from-time( )`, [2.0] `format-time( )`, [2.0] `minutes-from-dateTime( )`, and [2.0] `seconds-from-time( )` functions.

[2.0] tokenize()

Breaks a string into a sequence of strings, using a regular expression as a separator.

Syntax

```
xs:string* tokenize($input as xs:string?, $pattern as xs:string)
xs:string* tokenize($input as xs:string?, $pattern as xs:string,
                    $flags as xs:string)
```

Inputs

A string to be tokenized and a regular expression. The `tokenize( )` function also accepts a third string containing flags that change how the regular expression is evaluated.

It is a fatal error if a regular expression matches a zero-length string. See Appendix E for more details.

Outputs

A sequence of `xs:string`s, each of which represents a token parsed from the original string. The returned strings do not contain the separator.

Here are the full details of how `tokenize( )` works:

- If the first string is the empty sequence or a zero-length string, the empty sequence is returned.
- If the regular expression doesn't match anything in the input string, a singleton sequence containing the original input string is returned.
- If the regular expression matches the start of the string, the first string in the returned sequence will be an empty string (`""`). Similarly, if the regular expression matches the end of the string, the last string in the returned sequence will be an empty string.
- If the regular expression matches two overlapping strings in the input string, only the first match is replaced.
- The regular expression cannot be a zero-length string, nor can it match a zero-length string (in other words, `matches("", $pattern, $replacement)` can't be true). Assuming the input string is not of zero length, it *is* acceptable for a captured substring to be of zero length.

  If the expression `matches("", $pattern, $replacement)` is `true`, an error is raised. For example, using the pattern `'.?'` with any input string raises an error because this pattern matches a zero-length string.
- Regular expression matching does not use collations; the characters' Unicode code points are compared. Cases in which different characters are considered equal in the world's languages are not taken into account.
- Unlike regular expressions used in the `<xsl:analyze-string>` element, curly braces (`{` and `}`) are not doubled. (Curly braces used inside the `regex` attribute of

<xsl:analyze-string> must be doubled so they aren't interpreted as attribute value templates.)

- If a regular expression specifies more than one alternative, and more than one of those alternatives match at the same position in the input, the first alternative in the regular expression is the one that's used. To quote an example from the spec, `tokenize("abracadabra", "(ab)|(a)")` returns `("", "r", "c", "d", "r", "")`.

  The first alternative in the regular expression `(ab)` is considered the match in this situation, even though the second alternative `(a)` also matches.

- Finally, the third parameter modifies how the regular expression is processed. There are four different flags:

  `s`

  Regular expressions are evaluated in what the specs refer to as "dot-all" mode. When this flag is used, the dot operator (`.`) matches any character. Under normal processing (without the `s` flag), the dot operator matches any character *except* the newline character (`&#xA`). This flag is useful when you want to match strings that might include a newline character.

  `m`

  Regular expressions are evaulated in multiline mode. By default, the meta-character (`^`) matches the start of the entire string, while `$` matches the end of the entire string. In multiline mode, `^` matches the start of any line within the string, and `$` matches the end of any line within the string.

  `i`

  Regular expressions are evaluated in case-insensitive mode. The regular expression `"a"` matches both `"a"` and `"A"`.

  Note that Unicode issues can complicate this greatly. For example, the XQuery 1.0 and XPath 2.0 Functions and Operators spec gives the example of the Unicode sign for degrees Kelvin (`&#x212A`), which is the letter `"K"`. The combination of `regex="k"` and `flags="i"` matches the Kelvin sign as well as the letters `"k"` (`&#x6B`) and `"K"` (`&#x4B`).

  Other Unicode characters don't convert to letters. For example, the Unicode symbol for the Roman numeral `I` (`&#2160`) looks like the letter `I`, but does not convert to one. Fortunately, these complications are beyond the scope of this book.

  `x`

  All whitespace characters (`&#x9`, `&#xA`, `&#xD`, and `&#x20`) are removed from the regular expression before any comparison is done. In other words, with the `x` flag, the regular expressions `"John Smith"` and `"JohnSmith"` are the same. This flag is useful when you want to break a long regular expression into multiple lines to make it easier to read.

  The flags can be combined in any order. The parameters `'xis'` and `'six'` work exactly the same way.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 7.6, "String Functions that use Pattern Matching."

Example

Here's the stylesheet we'll use to illustrate how `tokenize( )` works:

```
<?xml version="1.0"?>
<!-- tokenize.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:variable name="string1"
      select="concat('Now is the time for all good men&#xA;',
              'and women &#xA;',
              'to aid the party.')"/>

    <xsl:variable name="string2" as="xs:string"
      select="'Visa # 1234-5678-9101-1121'"/>

    <xsl:text>&#xA;Here are some tests of the </xsl:text>
    <xsl:text>tokenize() function:&#xA;</xsl:text>

    <xsl:text>&#xA;  $string1 = &#xA;</xsl:text>
    <xsl:value-of select="$string1"/>

    <xsl:text>&#xA;&#xA;  $string2 = &#xA;</xsl:text>
    <xsl:value-of select="$string2"/>

    <xsl:text>&#xA;&#xA;  tokenize($string1, </xsl:text>
    <xsl:text>'&amp;#xA;') = </xsl:text>
    <xsl:text>&#xA;    ('</xsl:text>
    <xsl:value-of select="tokenize($string1, '&#xA;')"
      separator="',&#xA;     '"/>
    <xsl:text>')&#xA;    count() = </xsl:text>
    <xsl:value-of select="count(tokenize($string1, '&#xA;'))"/>

    <xsl:text>&#xA;&#xA;  tokenize($string2, '-') = </xsl:text>
    <xsl:text>&#xA;    ('</xsl:text>
    <xsl:value-of
      select="tokenize($string2, '-')" separator="', '"/>
    <xsl:text>')&#xA;    count() = </xsl:text>
    <xsl:value-of select="count(tokenize($string2, '-'))"/>

    <xsl:text>&#xA;&#xA;  tokenize($string2, </xsl:text>
    <xsl:text>'(Visa # )|-') = </xsl:text>
    <xsl:text>&#xA;    ('</xsl:text>
    <xsl:value-of
      select="tokenize($string2, '(Visa # )|-')" separator="', '"/>
```

```
    <xsl:text>')&#xA;    count() = </xsl:text>
    <xsl:value-of select="count(tokenize($string2, '(Visa #)|-'))"/>

  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Here are some tests of the tokenize() function:

  $string1 =
Now is the time for all good men
and women
to aid the party.

  $string2 =
Visa # 1234-5678-9101-1121

  tokenize($string1, '&#xA;') =
    ('Now is the time for all good men',
     'and women ',
     'to aid the party.')
    count() = 3

  tokenize($string2, '-') =
    ('Visa # 1234', '5678', '9101', '1121')
    count() = 4

  tokenize($string2, '(Visa # )|-') =
    ('', '1234', '5678', '9101', '1121')
    count() = 5
```

In the first test, we tokenized a string by its newline characters (`
`). Note that we don't have to use the `'s'` flag here because we're not looking for characters at the start or end of a line; we're simply looking for newline characters. For the second test, we broke our credit card number into four pieces. Unfortunately, the first token is `'Visa # 1234'`, which isn't what we wanted. To get just the four segments of numbers, we need to rewrite our regular expression as we did in test 3. Because the pattern `'Visa # '` matches the start of the string, the first token we get is a zero-length string.

The `count( )` function displayes how many tokens are in each sequence.

See Also

Appendix E has complete details on the way regular expressions work in XPath 2.0. Also see the definitions of the following elements and functions: [2.0] `<xsl:analyze-string>`, [2.0] `matches( )`, [2.0] `<xsl:matching-substring>`, [2.0] `<xsl:non-matching-substring>`, [2.0] `regex-group( )`, and [2.0] `replace( )`.

[2.0] trace()

Outputs diagnostic messages useful in tracing the processing of a stylesheet.

Syntax

```
item()* trace($value as item()*, $label as xs:string)
```

Inputs

A sequence of items and an `xs:string`. The string is typically something such as `The value of $x is:`. A diagnostic message containing that string and the value of the items in the sequence can be very useful in tracing the processing of the stylesheet.

Outputs

The output of the `trace( )` function is the input sequence of items. They are returned without changes.

In addition, `trace( )` converts the sequence of items into an `xs:string`. That string and the second argument are typically combined into a diagnostic message. *Almost all of the details of the* `trace( )` *function are implementation-dependent.* The format of the diagnostic message, the destination of the diagnostic message, and the order in which calls to the `trace( )` function are evaluated can vary from one processor to the next.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 4, "The Trace Function."

Example

This stylesheet is a smaller version of the stylesheet we used to illustrate the `data( )` function. We want to create trace messages each time we invoke the `datatest:print-item( )` function we created. Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- trace.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:datatest="http://www.oreilly.com">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:variable name="testSequence" as="item()*">
      <xsl:sequence select="(3)"/>
      <xsl:element name="currentDate">
        <xsl:value-of select="current-date()"/>
      </xsl:element>
      <xsl:element name="currentTime">
        <xsl:value-of select="current-time()"/>
      </xsl:element>
      <xsl:element name="integerTest">
        <xsl:value-of select="xs:integer(8)"/>
      </xsl:element>
      <xsl:sequence select="('blue', 'red')"/>
      <xsl:element name="floatTest1">
```

```
        <xsl:value-of select="xs:float(3.14)"/>
      </xsl:element>
      <xsl:element name="floatTest2">
        <xsl:value-of select="xs:float(42)"/>
      </xsl:element>
      <xsl:element name="dateTest">
        <xsl:value-of select="xs:date('1995-04-21')"/>
      </xsl:element>
    </xsl:variable>

    <xsl:text>&#xA;Here is a test of the </xsl:text>
    <xsl:text>trace() function:</xsl:text>

    <xsl:value-of
      select="for $i in (1 to count($testSequence))
              return (datatest:print-item(
              subsequence($testSequence, $i, 1)))"/>
  </xsl:template>

  <xsl:function name="datatest:print-item" as="xs:string">
    <xsl:param name="item" as="item()"/>
    <xsl:choose>
      <xsl:when test="$item instance of element()">
        <xsl:analyze-string regex="!!!.?"
          select="string(trace($item, 'trace()  element '))">
          <xsl:matching-substring></xsl:matching-substring>
        </xsl:analyze-string>
      </xsl:when>
      <xsl:otherwise>
        <xsl:analyze-string regex="!!!.?"
          select="string(trace($item, 'trace()  atom '))">
          <xsl:matching-substring></xsl:matching-substring>
        </xsl:analyze-string>
      </xsl:otherwise>
    </xsl:choose>
    <xsl:text> </xsl:text>
  </xsl:function>

</xsl:stylesheet>
```

Our stylesheet creates a sequence of items, and then calls our `datatest:print-item( )` function. When we invoke this with Saxon, we get these results:

```
trace()  atom : xs:integer: 3
trace()  element : element(currentDate, untyped): currentDate
trace()  element : element(currentTime, untyped): currentTime
trace()  element : element(integerTest, untyped): integerTest
trace()  atom : xs:string: blue
trace()  atom : xs:string: red
trace()  element : element(floatTest1, untyped): floatTest1
trace()  element : element(floatTest2, untyped): floatTest2
trace()  element : element(dateTest, untyped): dateTest

Here is a test of the trace() function:

3
```

```
2006-12-19-05:00
17:13:55-05:00
8
blue
red
3.14
42
1995-04-21
```

Notice that all of the `trace( )` messages are output before any of the `<xsl:text>` or `<xsl:value-of>` elements. The format of the messages and the order in which calls to `trace( )` are handled is implementation-dependent. Running the same stylesheet with the Altova XML engine produces a slightly different output:

```
trace()  atom 3
trace()  element <currentDate>2006-12-19-05:00</currentDate>
trace()  element <currentTime>17:30:28-05:00</currentTime>
trace()  element <integerTest>8</integerTest>
trace()  atom blue
trace()  atom red
trace()  element <floatTest1>3.14</floatTest1>
trace()  element <floatTest2>42</floatTest2>
trace()  element <dateTest>1995-04-21</dateTest>

Here is a test of the trace() function:

3
...
```

Again, all of the `trace( )` messages appear before the other output.

translate()

Allows you to convert individual characters in a string from one value to another. In many languages, this function is powerful enough to convert characters from one case to another. (*[2.0]* For case conversions, XQuery 1.0 and XPath 2.0 provide the more powerful `lower-case( )` and `upper-case( )` functions.)

Syntax

```
[1.0] string translate(string, $mapString as string, $transString as string)
[2.0] xs:string translate(xs:string?, $mapString as xs:string,
                          $transString as xs:string)
```

Inputs

Three strings. The first is the original, untranslated string, and the second and third strings define the characters to be converted.

Output

The original string, translated as follows:

- If a character in the original string appears in the second argument string (the mapping string), it is replaced with the corresponding character in the third argument string (the translation string).

  In other words, `translate('CAR', 'ABCDE', 'EXHQF')` returns `HER`. Going through the characters of the first string in order, `C` is the third character in the mapping string, so it is replaced with `H`, the third character in the translation string. The next character of the first string, `A`, is the first character in the mapping string, so it is replaced with `E`, the first character in the translation string. Finally, the last character of the first string, `R`, doesn't appear in the second string, so it is not modified at all.
- If a character in the original string appears in the second argument string and there is no corresponding character in the third argument string (the second argument string is longer than the third), then that character is deleted.

  In other words, `translate('CAR, 'ABCDER', 'EXHQF')` returns `HE`. The first two characters are translated as before. The last letter of the original string, `R`, is the sixth character of the mapping string. Because there is no sixth character in the translation string, the letter is deleted.
- If a character in the second argument string appears more than once, the first occurrence determines the replacement character.
- If the third argument string is longer than the second argument string, the extra characters are ignored.
- If the second string is a zero-length string, the original string is returned unchanged.
- *[2.0]* If the value of the first argument is the empty sequence, a zero-length string is returned.

Defined in

[1.0] XPath section 4.2, "String Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 7.4, "Functions on String Values."

Example

Here's a stylesheet with several examples of the `translate( )` function:

```
<?xml version="1.0"?>
<!-- translate.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Tests of the translate() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  Convert a string to uppercase:</xsl:text>
```

```
    <xsl:text>&#xA;    translate('Lily', </xsl:text>
    <xsl:text>'abcdefghijklmnopqrstuvwxyz', </xsl:text>
    <xsl:text>&#xA;              'ABCDEFGHIJKLMNOPQRSTUVWXYZ')=</xsl:text>
    <xsl:value-of
      select="translate('Lily', 'abcdefghijklmnopqrstuvwxyz',
                        'ABCDEFGHIJKLMNOPQRSTUVWXYZ')"/>
    <xsl:text>&#xA;&#xA;  Convert a string to lowercase:</xsl:text>
    <xsl:text>&#xA;    translate('Lily', </xsl:text>
    <xsl:text>'ABCDEFGHIJKLMNOPQRSTUVWXYZ', </xsl:text>
    <xsl:text>&#xA;              'abcdefghijklmnopqrstuvwxyz')=</xsl:text>
    <xsl:value-of
      select="translate('Lily', 'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
                        'abcdefghijklmnopqrstuvwxyz')"/>
    <xsl:text>&#xA;&#xA;  Remove parentheses, spaces, and dashes </xsl:text>
    <xsl:text>from a U.S. phone number:</xsl:text>
    <xsl:text>&#xA;    translate('(555) 555-1212', '() -', '')=</xsl:text>
    <xsl:value-of select="translate('(555) 555-1212', '() -', '')"/>
    <xsl:text>&#xA;&#xA;  Replace all but the last four digits of a </xsl:text>
    <xsl:text>credit card number with Xs:&#xA;</xsl:text>
    <xsl:variable name="credit" select="'1234 5678 9101 1810'"/>
    <xsl:text>    $credit='</xsl:text>
    <xsl:value-of select="$credit"/>
    <xsl:text>'</xsl:text>
    <xsl:text>&#xA;    translate(substring($credit, 1, 15), </xsl:text>
    <xsl:text>'1234567890 ', 'XXXXXXXXXX-')</xsl:text>
    <xsl:text>&#xA;    substring($credit, 16)</xsl:text>
    <xsl:text>&#xA;&#xA;    The first part is </xsl:text>
    <xsl:value-of
      select="translate(substring($credit, 1, 15), '1234567890 ',
                        'XXXXXXXXXX-')"/>
    <xsl:text>&#xA;    The second part is </xsl:text>
    <xsl:value-of select="substring($credit, 16)"/>
    <xsl:text>&#xA;    Here's how they look together: &#xA;      </xsl:text>
    <xsl:value-of
      select="translate(substring($credit, 1, 15), '1234567890 ',
                        'XXXXXXXXXX-')"/>
    <xsl:value-of select="substring($credit, 16)"/>
  </xsl:template>

</xsl:stylesheet>
```

When we use this stylesheet with any XML document, here are the results:

```
Tests of the translate() function:

  Convert a string to uppercase:
    translate('Lily', 'abcdefghijklmnopqrstuvwxyz',
              'ABCDEFGHIJKLMNOPQRSTUVWXYZ')=LILY

  Convert a string to lowercase:
    translate('Lily', 'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
              'abcdefghijklmnopqrstuvwxyz')=lily

  Remove parentheses, spaces, and dashes from a U.S. phone number:
    translate('(555) 555-1212', '() -', '')=5555551212
```

```
Replace all but the last four digits of a credit card number with Xs:
  $credit='1234 5678 9101 1810'
  translate(substring($credit, 1, 15), '1234567890 ', 'XXXXXXXXXX-')
  substring($credit, 16)

  The first part is XXXX-XXXX-XXXX-
  The second part is 1810
  Here's how they look together:
    XXXX-XXXX-XXXX-1810
```

true()

Always returns the boolean value `true`. Remember that the strings `"true"` and `"false"` don't have any special significance in XSLT. This function (and the `false( )` function) allow you to generate boolean values when you need them.

Syntax

[1.0] boolean **true()**
[2.0] xs:boolean **true()**

Inputs

None.

Output

The boolean value `true`.

Defined in

[1.0] XPath section 4.3, "Boolean Functions."

[2.0] XQuery 1.0 and XPath 2.0 Functions and Operators section 9.1, "Additional Boolean Constructor Functions."

Example

Here's a brief example that uses the `true( )` function:

```
<?xml version="1.0"?>
<!-- true.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the true() function:&#xA;&#xA;</xsl:text>

    <xsl:text>  true() returned </xsl:text>
    <xsl:value-of select="true()"/>
    <xsl:text>!</xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

When using this stylesheet against any XML document, it generates this less-than-exciting result:

```
A test of the true() function:

  true() returned true!
```

[2.0] type-available()

Given the name of a datatype, returns `true` if that datatype is known to the XSLT processor.

Syntax

```
xs:boolean type-available($typeName as xs:string)
```

Input

An `xs:string` containing the name of a datatype.

Output

The `xs:boolean` value `true` if the datatype is known to the XSLT processor; `false` otherwise.

Defined in

XSLT 2.0 section 18.1.4, "Testing Availability of Types."

Example

Here's a short stylesheet that checks the availability of two datatypes. The first datatype is defined to all XSLT 2.0 processors, while the second datatype is defined in a schema and therefore requires a schema-aware processor.

```
<?xml version="1.0"?>
<!-- type-available.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:age="http://www.oreilly.com/xslt"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the type-available() function:&#xA;</xsl:text>
    <xsl:text>&#xA;    xs:integer is available:  </xsl:text>
    <xsl:value-of select="type-available('xs:integer')"/>
    <xsl:text>&#xA;    age:age-type is available:  </xsl:text>
    <xsl:value-of select="type-available('age:age-type')"/>
  </xsl:template>

</xsl:stylesheet>
```

Notice that we have to declare the namespace prefixes for the datatypes. Because there is no `<xsl:import-schema>` element, only the first datatype is available:

```
A test of the type-available() function:

    xs:integer is available:  true
    age:age-type is available:  false
```

If we import a schema into the stylesheet, a schema-aware processor will find other types available:

```
<?xml version="1.0"?>
<!-- type-available2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:age="http://www.oreilly.com/xslt"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:import-schema namespace="http://www.oreilly.com/xslt">
    <xs:schema
      targetNamespace="http://www.oreilly.com/xslt"
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:simpleType name="age-type">
        <xs:restriction base="xs:integer">
          <xs:minInclusive value="0"/>
          <xs:maxInclusive value="130"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:schema>
  </xsl:import-schema>
  ...
</xsl:stylesheet>
```

With the schema imported, the stylesheet tells us the second datatype is available:

```
A test of the type-available() function:

    xs:integer is available:  true
    age:age-type is available:  true
```

[2.0] unordered()

Given a sequence, returns those items in an implementation-defined order.

Syntax

```
item()* unordered(item()*)
```

Inputs

A sequence of items.

Outputs

A sequence containing the same items as the input sequence, but in an order determined by the XSLT processor.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.1, "General Functions and Operators on Sequences."

Example

The `unordered( )` function is a way of telling the XSLT processor that we don't care how items are sequenced. In some cases, this can improve the performance of the XSLT processor. As an example, we'll look at the ancestors of an element in two ways. Using the normal approach, the ancestors appear in document order. When using `unordered( )` with Saxon, the ancestors appear in the reverse order. Most likely, Saxon is taking advantage of its internal data structures to start at a node and list its ancestors from the innermost level of the document outward. Using the same stylesheet with AltovaXML, the ancestors appear in document order in both cases.

We'll reuse one of our purchase orders as our input document:

```
<?xml version="1.0" ?>
<!-- po38293.xml -->
<purchase-order id="38293">
  <date year="2001" month="9" day="8"/>
  <customer id="4738" level="Basic">
    <address type="business">
      <name>
        <title>Ms.</title>
        <first-name>Amanda</first-name>
        <last-name>Reckonwith</last-name>
      </name>
      <street>930-A Chestnut Street</street>
      <city>Lynn</city>
      <state>MA</state>
      <zip>02930</zip>
    </address>
    <address type="ship-to"/>
  </customer>
  <items>
    <item part-no="23813-03-CDK">
      <name>Cucumber Decorating Kit</name>
      <qty>1</qty>
      <price>29.95</price>
    </item>
  </items>
</purchase-order>
```

We'll display the ancestors of the `<first-name>` element, selecting them once with a normal XPath expression and selecting them a second time with the `unordered( )` function. Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- unordered.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```
  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:text>&#xA;Here is a test of the unordered() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:for-each select="//first-name">
      <xsl:text>&#xA;  Element &lt;first-name&gt;</xsl:text>
      <xsl:value-of select="."/>
      <xsl:text>&lt;/first-name&gt;:&#xA;    </xsl:text>
      <xsl:text>&#xA;    Ancestors:&#xA;      </xsl:text>
      <xsl:value-of select="ancestor::*/name()" separator=", "/>
      <xsl:text>&#xA;&#xA;    Unordered ancestors:&#xA;      </xsl:text>
      <xsl:value-of select="unordered(ancestor::*/name())"
        separator=", "/>
    </xsl:for-each>

  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Here is a test of the unordered() function:

  Element <first-name>Amanda</first-name>:

    Ancestors:
      purchase-order, customer, address, name

    Unordered ancestors:
      name, address, customer, purchase-order
```

The first time we select the ancestors of the element, they appear in document order. The second time, the ancestors appear in what is presumably the most efficient way for Saxon to find them.

[2.0] unparsed-entity-public-id()

Returns the public ID of the unparsed entity with the specified name. If there is no such entity, `unparsed-entity-public-id( )` returns an empty string.

Syntax

```
xs:string unparsed-entity-public-id(xs:string)
```

Input

The name of the unparsed entity.

Output

The public ID of the unparsed entity with the specified name.

Defined in

XSLT 2.0 section 16.6, "Miscellaneous Additional Functions."

Example

Unparsed entities are rarely used; they typically refer to non-XML data, as in the entity `author-picture` defined in this XML document:

```
<?xml version="1.0"?>
<!-- unparsed-entity.xml -->
<!DOCTYPE book [
  <!ENTITY author-picture PUBLIC "-//OReilly//Author Images//DT"
    "dougtidwell.jpg" NDATA JPEG>
]>

<book>
  <prolog cover-image="author-picture"/>
  <body>
    <p>Pretend that lots of useful content appears here.</p>
  </body>
</book>
```

We'll use this stylesheet to return the public ID of our unparsed entity:

```
<?xml version="1.0"?>
<!-- unparsed-entity-public-id.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the unparsed-entity-</xsl:text>
    <xsl:text>public-id() function:</xsl:text>

    <xsl:text>&#xA;&#xA;   </xsl:text>
    <xsl:text>The public ID of the cover image is:&#xA;      </xsl:text>
    <xsl:value-of
      select="unparsed-entity-public-id(/book/prolog/@cover-image)"/>
    <xsl:text>.&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

When we transform the XML document with our stylesheet, the results look like this:

```
A test of the unparsed-entity-public-id() function:

   The public ID of the cover image is:
      -//OReilly//Author Images//DT.
```

unparsed-entity-uri()

Returns the URI of the unparsed entity with the specified name. If there is no such entity, `unparsed-entity-uri` returns an empty string.

Syntax

[1.0] string **unparsed-entity-uri(***string***)**
[2.0] xs:anyURI **unparsed-entity-uri(***xs:string***)**

Inputs

The name of the unparsed entity.

Output

The URI of the unparsed entity with the specified name.

Defined in

[1.0] XSLT section 12.4, "Miscellaneous Additional Functions."

[2.0] XSLT section 16.6, "Miscellaneous Additional Functions."

Example

Unparsed entities are rarely used; they typically refer to non-XML data, as in the entity `author-picture` in this XML document:

```
<?xml version="1.0"?>
<!-- unparsed-entity.xml -->
<!DOCTYPE book [
  <!ENTITY author-picture PUBLIC "-//OReilly//Author Images//DT"
    "dougtidwell.jpg" NDATA JPEG>
]>

<book>
  <prolog cover-image="author-picture"/>
  <body>
    <p>Pretend that lots of useful content appears here.</p>
  </body>
</book>
```

We'll use this stylesheet to get the public URI of our unparsed entity:

```
<?xml version="1.0"?>
<!-- unparsed-entity-uri.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;A test of the unparsed-entity-</xsl:text>
    <xsl:text>uri() function:</xsl:text>

    <xsl:text>&#xA;&#xA;   </xsl:text>
```

```
    <xsl:text>The URI of the cover image is:&#xA;      </xsl:text>
    <xsl:value-of
      select="unparsed-entity-uri(/book/prolog/@cover-image)"/>
    <xsl:text>.&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

When we transform the XML document with our stylesheet, the results look like this:

```
A test of the unparsed-entity-uri() function:

   The URI of the cover image is:
      file:/C:/projects/XSLTbookV2/AppendixC/dougtidwell.jpg.
```

The URI of the unparsed entity is based on the base URI of the XML document itself.

[2.0] unparsed-text()

Given a URI, returns the unparsed text of the resources identified by that URI.

Syntax

```
xs:string unparsed-text($href as xs:string)
xs:string unparsed-text($href as xs:string, $encoding as xs:string)
```

Inputs

An `xs:string` specifying the URI of the requested document. An optional second string specifies the document's encoding.

Output

An `xs:string` that contains the unparsed text of the URI.

Defined in

XSLT 2.0 section 16.2, "Reading Text Files."

Example

In this example, we'll generate an HTML page. Part of the page is a standard header and footer; those will be inserted into the document as unparsed text. If we use the `collection( )`, `doc( )`, or `document( )` functions, the data we read would have to be well-formed XML. Using `unparsed-text( )` instead, we can read documents that aren't necessarily well-formed (most HTML documents, for example) and use them.

We'll use our document of chocolate bar sales as the XML input document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- chocolate.xml -->
<report month="8" year="2006">
  <title>Chocolate bar sales</title>
  <brand>
    <name>Lindt</name>
```

```
    <units>27408</units>
  </brand>
  <brand>
    <name>Callebaut</name>
    <units>8203</units>
  </brand>
  <brand>
    <name>Valrhona</name>
    <units>22101</units>
  </brand>
  <brand>
    <name>Perugina</name>
    <units>14336</units>
  </brand>
  <brand>
    <name>Ghirardelli</name>
    <units>19268</units>
  </brand>
</report>
```

We'll generate an HTML report from this document. Our stylesheet uses `unparsed-text( )` to import an HTML header and footer:

```
<?xml version="1.0"?>
<!-- unparsed-text.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:variable name="reportTitle">
    <xsl:value-of select="/report/title"/>
    <xsl:text> for </xsl:text>
    <xsl:value-of select="/report/@month"/>
    <xsl:text>/</xsl:text>
    <xsl:value-of select="/report/@year"/>
  </xsl:variable>

  <xsl:template match="/">
    <html>
      <head>
        <title>
          <xsl:value-of select="$reportTitle"/>
        </title>
      </head>
      <body style="font-family: sans-serif;">
        <xsl:value-of
          select="unparsed-text('header.html')"
          disable-output-escaping="yes"/>
        <xsl:apply-templates select="*|text()"/>
        <xsl:value-of
          select="unparsed-text('footer.html')"
          disable-output-escaping="yes"/>
      </body>
    </html>
  </xsl:template>
```

```
  <xsl:template match="title">
    <h1>
      <xsl:value-of select="$reportTitle"/>
    </h1>
  </xsl:template>

  <xsl:template match="brand">
    <h2>
      <xsl:value-of select="name"/>
      <xsl:text> : </xsl:text>
      <xsl:value-of select="units"/>
    </h2>
  </xsl:template>

</xsl:stylesheet>
```

Notice that we used the `disable-output-escaping` attribute on `<xsl:value-of>`. Without this attribute, the markup we import is displayed as `<td>This header was generated...`. That means the text we import is treated as text, not as markup to be processed by the browser.

The document *header.html* looks like this:

```
<!-- header.html -->
<table width=33% border=1>
  <tr style="background: lightgray;">
    <td style="text-align: center;
               font-weight: bold;">
      This header was generated by<br>
      <code>unparsed-text()</code>, using a
      stylesheet from <br>O'Reilly's
      <cite>XSLT</cite>, 2nd edition.
    </td>
  </tr>
</table>
```

Notice that our HTML document features unquoted attributes on the `<table>` element and old-style `<br>` elements. Despite those violations of XML syntax, we can still import this document and use it in our results.

(With the exception of the word "footer" in place of "header," the file *footer.html* is exactly the same.)

Our completed document appears as in Figure C-6.

Another great use of the `unparsed-text( )` function is reading structured data formats such as comma-separated values. See the discussion in the section "[2.0] The unparsed-text() and unparsed-text-available() Functions" in Chapter 8 for a complete example.

[2.0] unparsed-text-available()

Given a URI, returns `true` if that document is available, `false` otherwise. This function allows you to check the existence and availability of a document before you attempt to open it with

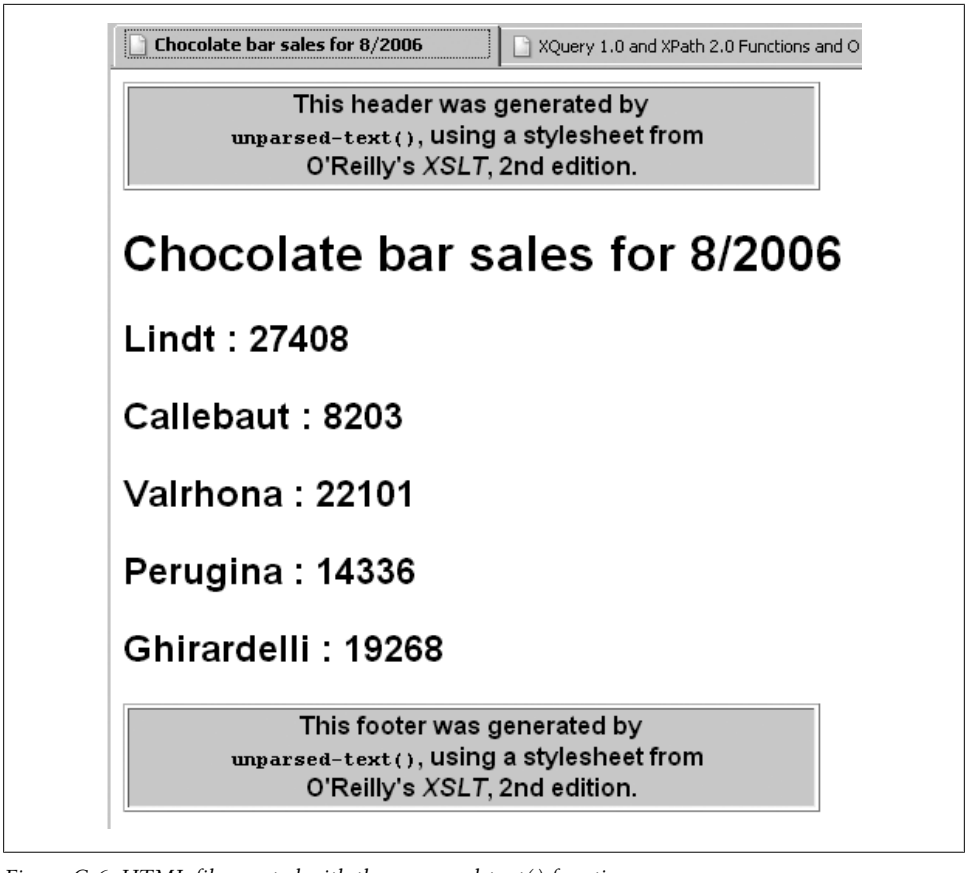

Chocolate bar sales for 8/2006
XQuery 1.0 and XPath 2.0 Functions and O
This header was generated by
unparsed-text(), using a stylesheet from
O'Reilly's XSLT, 2nd edition.
Chocolate bar sales for 8/2006
Lindt : 27408
Callebaut : 8203
Valrhona : 22101
Perugina : 14336
Ghirardelli : 19268
This footer was generated by
unparsed-text(), using a stylesheet from
O'Reilly's XSLT, 2nd edition.

Figure C-6. HTML file created with the unparsed-text( ) function

`unparsed-text( )`. If you attempt to open a document that is unavailable, the XSLT processor raises a fatal error.

Syntax

```
xs:boolean unparsed-text-available(xs:string)
xs:boolean unparsed-text-available(xs:string, xs:string)
```

Inputs

An `xs:string` specifying the URI of the requested document. An optional second string specifies the document's encoding.

Output

`true` if the document is unavailable; `false` otherwise.

Defined in

XSLT 2.0 section 16.2, "Reading Text Files."

Example

Here's a short stylesheet that checks to see whether three different files are available:

```
<?xml version="1.0"?>
<!-- unparsed-text-available.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:text>&#xA;Here are some tests of the </xsl:text>
    <xsl:text>unparsed-text-available() function:</xsl:text>

    <xsl:text>&#xA;&#xA;  unparsed-text-available</xsl:text>
    <xsl:text>('header.html') = </xsl:text>
    <xsl:value-of
      select="unparsed-text-available('header.html')"/>

    <xsl:text>&#xA;&#xA;  unparsed-text-available</xsl:text>
    <xsl:text>('disclaimer.html') = </xsl:text>
    <xsl:value-of
      select="unparsed-text-available('disclaimer.html')"/>

    <xsl:text>&#xA;&#xA;  unparsed-text-available</xsl:text>
    <xsl:text>('footer.html') = </xsl:text>
    <xsl:value-of
      select="unparsed-text-available('footer.html')"/>

  </xsl:template>

</xsl:stylesheet>
```

The results are:

```
Here are some tests of the unparsed-text-available() function:

  unparsed-text-available('header.html') = true

  unparsed-text-available('disclaimer.html') = false

  unparsed-text-available('footer.html') = true
```

The second file isn't available, but the first and third files are.

[2.0] upper-case()

Given a string, returns the uppercased version of that string.

Syntax

```
xs:string upper-case(xs:string?)
```

Inputs

An `xs:string` value.

Outputs

An `xs:string` in which all of the lowercase letters in the original string have been converted to uppercase. Any character that was originally in uppercase and any character that does not have an uppercase value is returned as is. If the value of the argument is the empty sequence, a zero-length string is returned.

Accented characters and other features of the world's languages mean that changing the case of a string might change its length. Also be aware that `upper-case( )` and `lower-case( )` are not always the inverse of each other in some languages. All of the case-conversion rules are defined by the Unicode standard, and XSLT processors are expected to conform with those rules.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 7.4, "Functions on String Values."

Example

Here is a stylesheet that illustrates the `upper-case( )` function. Notice that we're using `<xsl:output method="xml" encoding="UTF-8"/>` to make sure the character set is handled properly:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- upper-case.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" encoding="UTF-8" indent="yes"/>

  <xsl:template match="/">
    <testcase>
      <heading>Tests of the upper-case() function:</heading>
      <test>
        <label>upper-case('Lily') = </label>
        <result><xsl:value-of select="upper-case('Lily')"/></result>
      </test>
      <test>
        <label>upper-case('LILY') = </label>
        <result><xsl:value-of select="upper-case('LILY')"/></result>
      </test>
      <test>
        <label>upper-case('lily') = </label>
        <result><xsl:value-of select="upper-case('lily')"/></result>
      </test>
      <test>
        <label>uppercase('jalapeño') = </label>
        <result><xsl:value-of select="upper-case('jalapeño')"/></result>
      </test>
    </testcase>
  </xsl:template>
```

```
</xsl:stylesheet>
```

The results look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<testcase>
   <heading>Tests of the upper-case() function:</heading>
   <test>
      <label>upper-case('Lily') = </label>
      <result>LILY</result>
   </test>
   <test>
      <label>upper-case('LILY') = </label>
      <result>LILY</result>
   </test>
   <test>
      <label>upper-case('lily') = </label>
      <result>LILY</result>
   </test>
   <test>
      <label>uppercase('jalapeño') = </label>
      <result>JALAPEÑO</result>
   </test>
</testcase>
```

[2.0] year-from-date()

Given an `xs:date` value, returns its year component.

Syntax

```
xs:integer? year-from-date(xs:date?)
```

Input

An `xs:date` value.

Output

An `xs:integer` representing the year component of the given `xs:date` value. If the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `year-from-date( )` function:

```
<?xml version="1.0"?>
<!-- year-from-date.xsl -->
<xsl:stylesheet version="2.0"
```

```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xsl:output method="text"/>

    <xsl:template match="/">
      <xsl:text>&#xA;Extracting the year from an xs:date:</xsl:text>
      <xsl:variable name="currentDate" as="xs:date" select="current-date()"/>
      <xsl:text>&#xA;&#xA;  The current date is: </xsl:text>
      <xsl:value-of select="$currentDate"/>

      <xsl:text>&#xA;&#xA;  The current year: </xsl:text>
      <xsl:value-of select="year-from-date($currentDate)"/>
      <xsl:text>&#xA;    In words: </xsl:text>
      <xsl:value-of select="format-date($currentDate, '[YWw]')"/>
      <xsl:text>&#xA;    In German: </xsl:text>
      <xsl:value-of select="format-date($currentDate, '[YWw]', 'de', (), ())"/>
    </xsl:template>

</xsl:stylesheet>
```

The stylesheet creates these exciting results:

```
Extracting the year from an xs:date:

  The current date is: 2006-11-16-05:00

  The current year: 2006
    In words: Two Thousand and Six
    In German: Zweitausend Sechs
```

See Also

The definitions of the [2.0] `day-from-date( )`, [2.0] `format-date( )`, [2.0] `month-from-date( )`, and [2.0] `timezone-from-date( )` functions.

[2.0] year-from-dateTime()

Given an `xs:dateTime` value, returns its year value.

Syntax

```
xs:integer? year-from-dateTime(xs:dateTime?)
```

Inputs

An `xs:dateTime` value.

Output

An `xs:integer` representing the year component of the given `xs:dateTime` value. If the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `year-from-dateTime( )` function:

```
<?xml version="1.0"?>
<!-- year-from-datetime.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the year from an xs:dateTime:</xsl:text>
    <xsl:variable name="currentDateTime" as="xs:dateTime"
      select="current-dateTime()"/>
    <xsl:text>&#xA;&#xA;The current date and time is: </xsl:text>
    <xsl:value-of select="$currentDateTime"/>

    <xsl:text>&#xA;&#xA;  The current year: </xsl:text>
    <xsl:value-of select="year-from-dateTime($currentDateTime)"/>
    <xsl:text>&#xA;    In words: </xsl:text>
    <xsl:value-of select="format-dateTime($currentDateTime, '[YWw]')"/>
    <xsl:text>&#xA;    In German: </xsl:text>
    <xsl:value-of
      select="format-dateTime($currentDateTime, '[YWw]', 'de', (), ())"/>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet creates these results:

```
The current date and time is: 2006-11-16T05:18:20.187-05:00

  The current year: 2006
    In words: Two Thousand and Six
    In German: Zweitausend Sechs
```

See Also

The definitions of the [2.0] `day-from-dateTime( )`, [2.0] `format-dateTime( )`, [2.0] `hours-from-dateTime( )`, [2.0] `minutes-from-dateTime( )`, [2.0] `month-from-dateTime( )`, [2.0] `seconds-from-dateTime( )`, and [2.0] `timezone-from-dateTime( )` functions.

[2.0] years-from-duration()

Given an `xs:duration` value, returns the number of years in that duration.

Syntax

```
xs:integer? years-from-duration(xs:duration?)
```

Inputs

An `xs:duration` value.

Output

An `xs:integer` representing the years component of the given `xs:duration`. Be aware that for an `xs:dayTimeDuration`, this function always returns `0` because there is no years component of an `xs:dayTimeDuration`. Also, if the argument is the empty sequence, this function returns the empty sequence.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 10.5, "Component Extraction Functions on Durations, Dates and Times."

Example

This stylesheet demonstrates the `years-from-duration( )` function with all three types of durations:

```
<?xml version="1.0"?>
<!-- years-from-duration.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Extracting the years component from durations:</xsl:text>

    <xsl:variable name="sampleDuration" as="xs:duration"
      select="xs:duration('P3Y8M2DT4H23M12.2S')"/>
    <xsl:variable name="sampleYearMonthDuration" as="xs:yearMonthDuration"
      select="xs:yearMonthDuration('P3Y8M')"/>
    <xsl:variable name="sampleDayTimeDuration" as="xs:dayTimeDuration"
      select="xs:dayTimeDuration('P2DT4H23M12.2S')"/>

    <xsl:text>&#xA;&#xA;  A sample xs:duration: </xsl:text>
    <xsl:value-of select="$sampleDuration"/>
    <xsl:text>&#xA;    The years component of this duration is </xsl:text>
    <xsl:value-of select="years-from-duration($sampleDuration)"/>
    <xsl:text>.</xsl:text>

    <xsl:text>&#xA;&#xA;  A sample xs:yearMonthDuration: </xsl:text>
    <xsl:value-of select="$sampleYearMonthDuration"/>
    <xsl:text>&#xA;    The years component of this duration is </xsl:text>
    <xsl:value-of select="years-from-duration($sampleYearMonthDuration)"/>
    <xsl:text>.</xsl:text>

    <xsl:text>&#xA;&#xA;  A sample xs:dayTimeDuration: </xsl:text>
```

```
    <xsl:value-of select="$sampleDayTimeDuration"/>
    <xsl:text>&#xA;    The years component of this duration is </xsl:text>
    <xsl:value-of select="years-from-duration($sampleDayTimeDuration)"/>
    <xsl:text>.</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results from this stylesheet:

```
Extracting the years component from durations:

  A sample xs:duration: P3Y8M2DT4H23M12.2S
    The years component of this duration is 3.

  A sample xs:yearMonthDuration: P3Y8M
    The years component of this duration is 3.

  A sample xs:dayTimeDuration: P2DT4H23M12.2S
    The years component of this duration is 0.
```

As you can see, extracting the years component from an `xs:dayTimeDuration` returns `0`.

See Also

The definitions of the [2.0] `days-from-duration( )`, [2.0] `hours-from-duration( )`, [2.0] `minutes-from-duration( )`, [2.0] `months-from-duration( )`, and [2.0] `seconds-from-duration( )` functions.

[2.0] zero-or-one()

Raises an error unless its argument is a sequence containing zero or one items. For example, if a `<name>` element can have at most one `<title>` child, `zero-or-one(title)` would raise an error if a `<name>` element had the wrong number of `<title>` elements. Be aware that `zero-or-one( )` terminates processing; for a more flexible approach, use the `count( )` function to determine the cardinality of a sequence.

Syntax

```
item()? zero-or-one(item()*)
```

Inputs

A sequence of items. The sequence must either be empty or a singleton.

Outputs

This function returns the input sequence, assuming it has zero or one items. If the input sequence has more than one item, the `zero-or-one( )` function raises an error.

Defined in

XQuery 1.0 and XPath 2.0 Functions and Operators section 15.2, "Functions That Test the Cardinality of Sequences." More details about this function can be found in XQuery 1.0 and

XPath 2.0 Formal Semantics section 7.2, "Standard Functions with Specific Static Typing Rules."

Example

We'll look at a trivial example that illustrates the `zero-or-one( )` function. Invoking `zero-or-one( )` with a sequence containing more than one item raises an error. Here's the stylesheet:

```
<?xml version="1.0"?>
<!-- zero-or-one.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:output method="text"/>

  <xsl:template match="/">

    <xsl:variable name="emptySequence" as="item()*">
      <xsl:sequence select="()"/>
    </xsl:variable>

    <xsl:variable name="singleton" as="item()*">
      <xsl:sequence select="(3)"/>
    </xsl:variable>

    <xsl:text>&#xA;Here are two tests of the zero-or-one() </xsl:text>
    <xsl:text>function:&#xA;</xsl:text>

    <xsl:text>&#xA;  Calling zero-or-one() with the empty </xsl:text>
    <xsl:text>sequence:&#xA;    </xsl:text>

    <xsl:if test="count(zero-or-one($emptySequence) &lt; 2)">
      <xsl:text>&#xA;    Our sequence has zero or one items!</xsl:text>
    </xsl:if>

    <xsl:text>&#xA;&#xA;  Calling zero-or-one() with a </xsl:text>
    <xsl:text>singleton:&#xA;</xsl:text>
    <xsl:text>&#xA;    Our sequence is:&#xA;      </xsl:text>
    <xsl:value-of select="$singleton" separator="&#xA;    "/>

    <xsl:if test="count(zero-or-one($singleton) &lt; 2)">
      <xsl:text>&#xA;&#xA;    Our sequence has zero or one items!</xsl:text>
    </xsl:if>
  </xsl:template>

</xsl:stylesheet>
```

Notice that we use the `count( )` function to create a boolean value here. Converting a sequence into a boolean value directly won't work here. If the sequence is the empty sequence, the XSLT 2.0 processor evaluates that as `false`, which is the opposite of what we want. Even if the sequence has one member, there are still problems; the sequence `(0)` evaluates as `false`, and the sequence `(xs:date('1995-04-21'))` raises an error. The best way to get a boolean value here is to see whether the size of the returned sequence is `< 2`.

The stylesheet generates these results:

```
Here are two tests of the zero-or-one() function:

  Calling zero-or-one() with the empty sequence:

    Our sequence has zero or one items!

  Calling zero-or-one() with a singleton:

    Our sequence is:
      3

    Our sequence has zero or one items!
```

Calling `zero-or-one( )` with the empty sequence or a singleton returns the input sequence; calling `zero-or-one( )` with anything else raises an error.

See Also

The descriptions of the `count( )`, [2.0] `empty( )`, [2.0] `exactly-one( )`, and the [2.0] `one-or-more( )` functions and the discussion of the XPath 2.0 `treat as` operator in the section "[2.0] Datatype Operators—instance of, castable as, cast as, and treat as" in Chapter 3.

APPENDIX D
XML Schema Overview

This appendix is a brief overview of XML Schema. For in-depth information about XML Schemas, I highly recommend Eric van der Vlist's *XML Schema*, published by O'Reilly. The *XML Schema Primer* at the W3C is very useful as well.

We'll cover three topics in this appendix. First, we'll look at how to declare elements and attributes in XML Schema. Next we'll go through the many ways to create types. Finally, we'll look at how to use XML Schemas in XSLT stylesheets. The focus here is on the aspects of XML Schema that apply to XSLT. A schema-aware XSLT processor can use a schema to validate a document or element. If we're working with types declared in an XML Schema, a schema-aware processor can use those datatypes just like `xs:string`, `xs:date`, or any other basic datatype.

Declaring Elements and Attributes

The most common task in XML Schema is declaring elements and attributes. We'll start with an empty element, then move on to more sophisticated things.

Creating an Empty Element

Here's how we create an empty element:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- empty1.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="empty1">
    <xs:complexType/>
  </xs:element>

</xs:schema>
```

That's it. The empty <xs:complexType> means that our element can't have any attributes and it can't have any content. It's not terribly useful, but that's how it works. An XML document that uses this schema looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- empty1.xml -->
<empty1
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt empty1.xsd"/>
```

Although we said our empty element couldn't contain attributes, it looks like we have three of them here. The first two (xmlns and xmlns:xsi) are actually namespace declarations, not attributes. The xsi:schemaLocation attribute (it actually *is* an attribute) associates the XML Schema with our document. In the schema, the targetNamespace attribute defines the namespace used by this schema; it's also the default namespace (xmlns="http://www.oreilly.com/xslt") in the schema and the XML document. To tie everything together, the xsi:schemaLocation attribute in our XML document tells the XML parser where to find the schema that defines what a valid document looks like.

There are other ways of using namespaces, but we won't cover them in any detail here. For example, our XML document could look like this:

```
<ora:empty1
  xmlns:ora="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt empty1.xsd"/>
```

We took out the default namespace and declared a namespace prefix that matches the namespace in the XML Schema. Because the XML document doesn't have a default namespace, we have to qualify all of the elements with the namespace prefix that matches the target namespace of the schema. That's why our element is now <ora:empty1> instead of <empty1>.

Creating an Empty Element with Attributes

Our next step is to create an attribute for our empty element. To do that, we declare what XML Schema refers to as a *complex type*:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- empty2.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="empty2">
    <xs:complexType>
      <xs:attribute name="color"/>
    </xs:complexType>
  </xs:element>
```

```
</xs:schema>
```

The XML document is slightly more interesting now:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- empty2.xml -->
<empty2
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt empty2.xsd"
  color="blue"/>
```

We simply declared an attribute with a name here. We could have declared a datatype for the attribute; we'll do that soon.

Creating an Element with Text

To create an element that's actually useful, we'll want to let it contain something. We'll start by simply creating an element whose type is `xs:string`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- content1.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="content1" type="xs:string"/>

</xs:schema>
```

Notice that we didn't use `<xs:complexType>` this time. We simply said our element was a string. Here's a valid document for this schema:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- content1.xml -->
<content1
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt content1.xsd">
Our element now contains some text!  It's getting more useful
all the time.
</content1>
```

Creating an Element with Text and Attributes

Now we'll create an element that has both text and attributes. Remember, in order to create attributes, we have to create an `<xs:complexType>`, so we have to create a new type based on `xs:string`. Here's how we do that:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- content2.xsd -->
<xs:schema
```

```
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="content2">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="color" type="xs:string"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Our new element is a complex type, but it uses what XML Schema refers to as *simple content*. In other words, this element is an extension of a simple type, `xs:string`. The extension to the simple type is that we're adding an attribute. Here's how our document looks now:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- content2.xml -->
<content2
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt content2.xsd"
  color="blue">
Our element now contains some text!  It's getting more useful
all the time.
</content2>
```

Creating an Element with Mixed Content

For our final example, we'll create an element with mixed content. *Mixed content* means this element can contain text and other elements. Here's an expanded schema that declares a second element; the first element can contain any combination of text and the second element:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- content3.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="content3">
    <xs:complexType mixed="true">
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="emphasis"/>
      </xs:sequence>
      <xs:attribute name="color" type="xs:string"/>
    </xs:complexType>
  </xs:element>
```

```
    <xs:element name="emphasis" type="xs:string"/>

</xs:schema>
```

Now we've declared a complex type that contains mixed content, an element, and an attribute. Here's a valid document in our new schema:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- content3.xml -->
<content3
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt content3.xsd"
  color="blue">
Our element <emphasis>now</emphasis> contains some text!
It's getting <emphasis>more useful</emphasis> all the time.
</content3>
```

One final detail: in the schema, we declared the `<emphasis>` element outside the complex type and referred to it in the declaration of the `<content3>` element. If we had declared `<emphasis>` like this:

```
<xs:complexType mixed="true">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:element name="emphasis" type="xs:string"/>
    ...
```

we would get the same results, but we couldn't reuse the `<emphasis>` element anywhere else. In all our examples from now on, we'll declare elements globally and reference them where we need them. (A *globally declared element* is an `<xs:element>` whose parent is the `<xs:schema>` element.)

Defining Datatypes

Some of the elements and attributes we've created to this point use the built-in XML Schema datatypes. The ability to define our own types is a powerful feature of the language. Custom datatypes are the feature most directly related to XSLT. With a schema-aware processor, we can use our own datatypes for validation. We'll cover the ways to create datatypes in the next section.

We don't cover the basic datatypes (`xs:string`, `xs:integer`, etc.) used by XML Schema here. See the discussion of "*[2.0]* XPath 2.0 Datatypes" in Appendix B for all the details on those datatypes.

Anonymous Types

Everything we've done to this point has used anonymous types. That means we used `<xs:simpleType>` or `<xs:complexType>`, but we didn't give those types a name. That was

OK for our simple schemas because we didn't want to reuse those datatypes outside the element in which they were defined. From now on, we'll give our datatypes a name so we can use them whenever we need to. Here's the difference between an anonymous type and a named type:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- content4.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="content4a">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="color" type="xs:string"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="content4b-type">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="color" type="xs:string"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:element name="content4b" type="content4b-type"/>

</xs:schema>
```

The first element in the schema, `<content4a>`, uses an anonymous type. The second element has the exact same structure, but it uses the named datatype we created. The difference, of course, is that we can use the named datatype anywhere, while the anonymous type exists only inside the declaration of element `<content4a>`.

Groups

There are three kinds of groups we can define in an XML Schema: `<xs:sequence>` groups, `<xs:choice>` groups, and `<xs:all>` groups. Normally these groups are inside the declaration of a type or element, but we can also use the `<xs:group>` element to create a group separately and refer to it as we need it.

The most flexible group is a choice group. A *choice group* contains a list of elements, only one of which may appear in a valid XML document. Although that sounds restrictive, we can say that a choice group can appear zero or more times, and we can put it in a mixed content model. For example, here's a choice group that says a `<p>` element can contain text and any combination of `<a>`, `<b>`, `<br>`, `<code>`, `<i>`, or `<img>` elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- paragraph.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="a">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="href" type="xs:string"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>

  <xs:element name="b">
    <xs:complexType mixed="true">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="a"/>
        <xs:element ref="br"/>
        <xs:element ref="code"/>
        <xs:element ref="i"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>

  <xs:element name="br"/>

  <xs:element name="code">
    <xs:complexType mixed="true">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="a"/>
        <xs:element ref="b"/>
        <xs:element ref="br"/>
        <xs:element ref="i"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>

  <xs:element name="i">
    <xs:complexType mixed="true">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="a"/>
        <xs:element ref="b"/>
        <xs:element ref="br"/>
        <xs:element ref="code"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>

  <xs:element name="img">
    <xs:complexType>
      <xs:attribute name="href" type="xs:string"/>
```

```
    </xs:complexType>
  </xs:element>

  <xs:element name="p">
    <xs:complexType mixed="true">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="a"/>
        <xs:element ref="b"/>
        <xs:element ref="br"/>
        <xs:element ref="code"/>
        <xs:element ref="i"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

All of the elements that have mixed content can also contain any number of elements from the `<xs:choice>`. This schema gives us a great deal of flexibility; we can have a `<p>` element that contains text that contains a `<code>` element, which in turn contains a `<b>` element, which in turn contains an `<i>` element.

Next we'll look at sequence groups. A sequence group is defined with `<xs:sequence>` and contains a list of elements that must appear in the sequence in which they are listed. We can `minOccurs="0"` to indicate that some of the elements are optional. Here's a schema that defines elements that must appear inside a `<person>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- person.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="age"/>
        <xs:element ref="birthday"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="name" type="xs:string"/>
  <xs:element name="age" type="xs:positiveInteger"/>
  <xs:element name="birthday" type="xs:date"/>

</xs:schema>
```

This schema says that any `<person>` element must contain a `<name>` element, an `<age>` element, and a `<birthday>` element. The elements must occur in this order, and all of them are required.

The final kind of group we'll look at is the <xs:all> group. An <xs:all> group contains a list of elements. All of the elements in the list must appear once, although the order in which they appear doesn't matter. We can define some of the elements as being optional by using minOccurs="0" if necessary. The only legal values for minOccurs and maxOccurs are 0 and 1. Here's a schema with an <xs:all> group:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- person-all.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="person">
    <xs:complexType>
      <xs:all>
        <xs:element ref="name"/>
        <xs:element ref="age"/>
        <xs:element ref="birthday"/>
      </xs:all>
    </xs:complexType>
  </xs:element>

  <xs:element name="name" type="xs:string"/>
  <xs:element name="birthday" type="xs:date" nillable="true"/>
  <xs:element name="age" type="xs:positiveInteger"/>

</xs:schema>
```

The three elements here can occur in any order, but all three must occur. Here's an XML document that's valid against this schema:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- person-all.xml -->
<person
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt person-all.xsd">
  <age>42</age>
  <birthday>1965-06-19</birthday>
  <name>Doug Tidwell</name>
</person>
```

There is one important restriction on <xs:all> groups. The elements defined or referenced inside the group can only be individual elements. You can't use a group to add multiple elements to the group—you have to do that individually.

One more thing before we leave the topic of groups: XML Schema defines the <xs:group> element. We can use it to define a group of elements or to reference a group of elements defined elsewhere. Here's an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- group.xsd -->
<xs:schema
```

```
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="person">
    <xs:complexType>
      <xs:group ref="person-elements"/>
    </xs:complexType>
  </xs:element>

  <xs:group name="person-elements">
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="birthday"/>
      <xs:element ref="age"/>
    </xs:sequence>
  </xs:group>

  <xs:element name="name" type="xs:string"/>
  <xs:element name="birthday" type="xs:date"/>
  <xs:element name="age" type="xs:positiveInteger"/>

</xs:schema>
```

We define a named group and refer to it in the definition of the `<person>` element.

Creating New Datatypes by Restriction

A common way to create new datatypes is to restrict the values of another datatype. We'll look at three approaches: ranges, enumerations, and regular expressions.

For our first example, we'll use a range to define a datatype called `age` based on `xs:integer`. We would like for the `age` datatype to have a value between `0` and `130`. (We're being optimistic about longevity here.) Here's how we define a range of valid values:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- person1.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="age"/>
      </xs:sequence>
      <xs:attribute name="eyeColor" type="xs:string"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="name" type="xs:string"/>
```

```
  <xs:simpleType name="age-type">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="130"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:element name="age" type="age-type"/>

</xs:schema>
```

We've defined a new datatype, age-type, and we've used it as the datatype for the element <age>. Here's a valid document for this schema:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- person1.xml -->
<person
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt person1.xsd"
  eyeColor="brown">
  <name>Doug Tidwell</name>
  <age>42</age>
</person>
```

If the value of <age> is outside the range we defined, this document won't validate.

We'll continue developing our person schema by limiting the value of the eyeColor attribute. For simplicity's sake, we'll limit this attribute to four colors. We do that with an enumeration:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- person2.xsd -->
...
      <xs:attribute name="eyeColor" type="eyeColor-type"/>
    </xs:complexType>
  </xs:element>

  <xs:simpleType name="eyeColor-type">
    <xs:restriction base="xs:string">
      <xs:enumeration value="blue"/>
      <xs:enumeration value="brown"/>
      <xs:enumeration value="gray"/>
      <xs:enumeration value="green"/>
    </xs:restriction>
  </xs:simpleType>
  ...
```

If the value of the eyeColor attribute is anything other than the four values defined here, validation fails.

For a final use of restrictions, we'll use a regular expression. The regular expression defines what valid data looks like; if a value of that type doesn't match the regular expression, validation fails. Here's a datatype defined with a regular expression:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- person3.xsd -->
...
  <xs:element name="driversLicense" type="driversLicense-type"/>
...
  <xs:simpleType name="driversLicense-type">
    <xs:restriction base="xs:string">
      <xs:pattern value="[A-Z]{2}-[0-9]{4}-[0-9]{6}"/>
    </xs:restriction>
  </xs:simpleType>
  ...
```

Here's a document that's valid according to this schema:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- person3.xml -->
<person
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt person3.xsd"
  eyeColor="brown">
  <name>Doug Tidwell</name>
  <age>42</age>
  <driversLicense>NC-3821-388297</driversLicense>
</person>
```

(Notice that we defined the element that uses the datatype before the datatype itself. The order in which you define things in the schema doesn't matter.)

Because we've defined our own datatypes through restriction, validation fails if `<age>` is outside the defined range, *or* `eyeColor` isn't one of our four colors, *or* `<driversLicense>` doesn't match our regular expression. Because all of these datatypes are named, we can reuse them wherever we need them.

Creating New Datatypes by Extension

Another way to create new datatypes is through extension. The most common example of this is adding an attribute to a simple type. The XML Schema spec says that simple types (`xs:string`, `xs:integer`, etc.) can't have attributes. We can create a complex type that allows an element to have content of type `xs:integer` and have an attribute. Here's how the schema looks:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- extension.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:simpleType name="currency-type">
    <xs:restriction base="xs:string">
      <xs:enumeration value="USD"/>
      <xs:enumeration value="GBP"/>
```

```
      <xs:enumeration value="CNY"/>
      <xs:enumeration value="EUR"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="price-type">
    <xs:simpleContent>
      <xs:extension base="xs:decimal">
        <xs:attribute name="currency" type="currency-type"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:element name="price" type="price-type"/>

</xs:schema>
```

Now we have an element whose content must be an `xs:decimal`, while it must also have an attribute named `currency`. Here's a valid document for this schema:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- extension.xml -->
<price
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt extension.xsd"
  currency="USD">
438.92
</price>
```

If we change the content of the element to be `akd482.58`, validation fails.

Casting Between Datatypes

It is possible to cast an atomic value from one atomic datatype to another. We can't do that with complex types, but it does work for atomic types. Here's a short example that illustrates this:

```
<?xml version="1.0"?>
<!-- typecasting.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:age="http://www.oreilly.com/xslt"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:import-schema namespace="http://www.oreilly.com/xslt">
    <xs:schema
      targetNamespace="http://www.oreilly.com/xslt"
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:simpleType name="age-type">
        <xs:restriction base="xs:integer">
          <xs:minInclusive value="0"/>
          <xs:maxInclusive value="130"/>
```

```
        </xs:restriction>
      </xs:simpleType>
    </xs:schema>
  </xsl:import-schema>

  <xsl:template match="/">
    <xsl:variable name="age" as="age:age-type"
      select="age:age-type(42)"/>
    <xsl:variable name="age-int" as="xs:integer"
      select="$age cast as xs:integer"/>
    <xsl:variable name="float-age" as="age:age-type"
      select="xs:float(42.0) cast as age:age-type"/>
    <xsl:value-of select="$age, $age-int, $float-age"
      separator=", "/>
  </xsl:template>
</xsl:stylesheet>
```

The three variables at the bottom of the stylesheet are of datatypes `age:age-type`, `xs:integer`, and `xs:float`. Because all of these are numeric types, we can cast a value from one type to another. We still have to follow the restrictions for each datatype; the statement `xs:float(148.3) cast as age:age-type` fails at runtime. Casting `xs:float(48.3)` to an `age:age-type` creates a new `age:age-type` value of `48`.

To "cast" to or from a complex type, we have to build the complex type ourselves. In other words, to create a new complex type, we have to use `<xsl:element>` to create and validate the complex type, filling in the elements and attributes of the complex type appropriately. See the second example of the `<xsl:element>` element in Appendix A for more details.

Creating List Types

Another datatype we can create is a list type. A list type references a datatype; the new list type allows space-separated values of that type. We'll add a new datatype called `state-abbr` to our schema, then create a list type based on it. Here's the schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- list.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="birthday"/>
        <xs:element ref="age"/>
        <xs:element ref="priorAddresses"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

```
  <xs:simpleType name="state-abbr-type">
    <xs:restriction base="xs:string">
      <xs:length value="2"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="state-abbr-list-type">
    <xs:list itemType="state-abbr-type"/>
  </xs:simpleType>

  <xs:element name="name" type="xs:string"/>
  <xs:element name="birthday" type="xs:date"/>
  <xs:element name="age" type="xs:positiveInteger"/>
  <xs:element name="priorAddresses">
    <xs:complexType>
      <xs:attribute name="states" type="state-abbr-list-type"/>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Now we have an attribute named `states` whose content is a list of one or more state abbreviations. Here's a valid document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- list.xml -->
<person
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt list.xsd">
  <name>Doug Tidwell</name>
  <birthday>1965-06-19</birthday>
  <age>42</age>
  <priorAddresses states="GA TN NC"/>
</person>
```

Creating Union Types

A union type references two or more datatypes. The valid content of a union type is a value from one of those datatypes. We'll create another schema that defines a list of zipcodes, then create a union type that allows the `states` attribute to be either a list of state abbreviations or a list of zipcodes. Here's an excerpt from the schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- union.xsd -->
...
  <xs:simpleType name="zipcode">
    <xs:restriction base="xs:integer">
      <xs:pattern value="[0-9]{5}"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="zipcode-list-type">
```

```
    <xs:list itemType="zipcode"/>
  </xs:simpleType>

  <xs:simpleType name="state-abbr-type">
    <xs:restriction base="xs:string">
      <xs:length value="2"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="state-abbr-list-type">
    <xs:list itemType="state-abbr-type"/>
  </xs:simpleType>

  <xs:simpleType name="address-union-type">
    <xs:union memberTypes="state-abbr-list-type zipcode-list-type"/>
  </xs:simpleType>
  ...
  <xs:element name="priorAddresses">
    <xs:complexType>
      <xs:attribute name="states" type="address-union-type"/>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

And here's a valid document for this schema:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- union.xml -->
<person
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt union.xsd">
  <name>Doug Tidwell</name>
  <birthday>1965-06-19</birthday>
  <age>42</age>
  <priorAddresses states="27516 37174 30606"/>
</person>
```

A `states` attribute that contains a list of zip codes or a list of state abbreviations is legal. What we *can't* do is combine the two. The following is illegal:

```
  <!-- Not valid! -->
  <priorAddresses states="27516 37174 30606 NC TN GA"/>
```

Substitution Groups

A substitution group is a set of elements that can be substituted for each other. We'll add an `<address>` element to our schema, then we'll define two types of address that can be used instead. Here's the schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- substitute.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
```

```
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="birthday"/>
        <xs:element ref="age"/>
        <xs:element ref="address"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="name" type="xs:string"/>
  <xs:element name="birthday" type="xs:date"/>
  <xs:element name="age" type="xs:positiveInteger"/>

  <xs:element name="address" type="xs:string"/>
  <xs:element name="businessAddress" type="xs:string"
    substitutionGroup="address"/>
  <xs:element name="residentialAddress" type="xs:string"
    substitutionGroup="address"/>

</xs:schema>
```

The schema defines two additional elements, `<businessAddress>` and `<residentialAddress>`, that can be substituted for `<address>`. Notice that the two additional elements use the `substitutionGroup` attribute to reference the element they can replace. Also, all of the elements are the same type. Here's a valid XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- substitute.xml -->
<person
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt substitute.xsd">
  <name>Doug Tidwell</name>
  <birthday>1965-06-19</birthday>
  <age>42</age>
  <address>4013 Corporate Parkway</address>
</person>
```

This document is valid as well:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- substitute2.xml -->
<person
...
  <businessAddress>4013 Corporate Parkway</businessAddress>
</person>
```

And so is this:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- substitute3.xml -->
```

```
<person
...
  <residentialAddress>1234 Main Street</residentialAddress>
</person>
```

Abstract Elements and Datatypes

Our final topic is the use of *abstract elements and datatypes*. An abstract element or datatype works like an abstract class in object-oriented programming languages. We define the abstract class, but we have to create a subclass to actually use it. In terms of XML Schema, we'll be creating new elements and datatypes based on the abstract ones.

In our first example, we'll define an abstract element and its properties, and then use our substitution group as before. The only change to the schema is that we've added `abstract="true"` to the definition of the `<address>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- abstract1.xsd -->
<xs:schema
...
  <xs:element name="address" type="xs:string" abstract="true"/>

  <xs:element name="businessAddress" type="xs:string"
    substitutionGroup="address"/>
  <xs:element name="residentialAddress" type="xs:string"
    substitutionGroup="address"/>
...
```

The effect of the abstract element is that we're forcing the document to contain either a `<businessAddress>` or a `<residentialAddress>`. This document is valid:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- abstract1.xml -->
<person
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt abstract1.xsd">
  <name>Doug Tidwell</name>
  <birthday>1965-06-19</birthday>
  <age>42</age>
  <businessAddress>4013 Corporate Parkway</businessAddress>
</person>
```

If this document used the `<address>` element instead of `<businessAddress>`, it would not be valid.

Finally, we'll look at an abstract datatype. In this example, we'll define an abstract datatype named `state-or-province-type` and an element of that type named `<state-or-province>`. We'll then define two nonabstract extensions of those types called `province-type` and `state-type`. The `<xsl:state-or-province>` elements in our sample document will use the `xsi:type` attribute to declare the element to be of a nonabstract type:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- abstract2.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="birthday"/>
        <xs:element ref="age"/>
        <xs:element ref="state-or-province"
          minOccurs="2" maxOccurs="2"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="name" type="xs:string"/>
  <xs:element name="birthday" type="xs:date"/>
  <xs:element name="age" type="xs:positiveInteger"/>

  <xs:element name="address" type="xs:string"/>

  <xs:complexType name="state-or-province-type" abstract="true"/>

  <xs:complexType name="state-type">
    <xs:complexContent>
      <xs:extension base="state-or-province-type">
        <xs:attribute name="state-abbr">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="NC"/>
              <xs:enumeration value="TN"/>
              <xs:enumeration value="GA"/>
              <!-- Other states left out -->
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="province-type">
    <xs:complexContent>
      <xs:extension base="state-or-province-type">
        <xs:attribute name="province-abbr">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="NS"/>
              <xs:enumeration value="BC"/>
              <xs:enumeration value="PEI"/>
              <!-- Other provinces left out -->
            </xs:restriction>
```

```
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:element name="state-or-province" type="state-or-province-type"/>

</xs:schema>
```

The two datatypes we define extend the abstract datatype by adding an attribute. Each attribute is restricted with an enumeration. The `state-type` datatype has a `state-abbr` attribute that can only contain values from our enumeration; the same is true for the `province-type` datatype and its `province-abbr` attribute.

Here's a valid XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- abstract2.xml -->
<person
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt abstract2.xsd">
  <name>Doug Tidwell</name>
  <birthday>1965-06-19</birthday>
  <age>42</age>
  <state-or-province xsi:type="province-type" province-abbr="NS"/>
  <state-or-province xsi:type="state-type" state-abbr="NC"/>
</person>
```

(For the sake of illustration, we changed the `minOccurs` attribute on the `<state-or-province>` element so we could use both nonabstract datatypes here.) Because the `<state-or-province>` element is declared with an abstract datatype, we have to use the `xsi:type` attribute to declare a nonabstract datatype for the element.

Using an XML Schema in a Stylesheet

Now that we've covered how schemas work, we'll take a look at how to use them in XSLT. The `<xsl:import-schema>` element lets us import an XML Schema into our stylesheet directly. In addition to `<xsl:import-schema>`, an XML document can reference a schema with the `xsi:noNamespaceSchemaLocation` or `xsi:schemaLocation` attributes. If our stylesheet is set up correctly, the XSLT processor can use the referenced schema to validate data.

Importing XML Schemas with <xsl:import-schema>

The `<xsl:import-schema>` element allows you to import an XML Schema. The schema is imported and processed before any input documents are processed. This allows you to define datatypes and validation rules before the XSLT processor begins to transform

the input document. *This element is only supported by schema-aware XSLT 2.0 processors.*

The element has two optional parameters: `namespace`, which defines the namespace URI for the schema, and `schema-location`, which contains the URI of the schema file itself. `<xsl:import-schema>` can use `schema-location` to import a file, or it can contain the actual XML Schema within the stylesheet. We'll look at a couple of examples here; for more complete examples, see the discussion of the [2.0 – Schema] `<xsl:import-schema>` element.

The simplest way to import a schema into a stylesheet is to use the URI:

```
<?xml version="1.0"?>
<!-- import-schema.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:po="http://www.oreilly.com/xslt">

  <xsl:import-schema namespace="http://www.oreilly.com/xslt"
    schema-location="po.xsd" />

  <xsl:output method="text"/>

  <xsl:template match="schema-element(po:purchase-order)">
    <xsl:text>&#xA;This is a test of the &lt;xsl:import-</xsl:text>
    <xsl:text>schema&gt; element.&#xA;&#xA;</xsl:text>
    <xsl:text>Here are all the items in this purchase </xsl:text>
    <xsl:text>order:&#xA;</xsl:text>
    <xsl:for-each select="po:items/po:item">
      <xsl:text>  * </xsl:text>
      <xsl:value-of select="po:partname"/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>

  <xsl:template match="*">
    <xsl:message terminate="yes">
      <xsl:text>This is not a valid purchase order!</xsl:text>
    </xsl:message>
  </xsl:template>

</xsl:stylesheet>
```

Notice that the imported schema is associated with a namespace URI, which in turn is associated with the prefix `po`. The XSLT pattern uses the `schema-element(po:purchase-order)` node test to find all of the elements named `<purchase-order>` with a namespace URI of `http://www.oreilly.com/xslt`. Given that starting point, the stylesheet finds all of the `<item>` elements in the purchase order. The output looks like this:

```
This is a test of the <xsl:import-schema> element.

Here are all the items in this purchase order:
```

```
  * Turnip Twaddler
  * Clam Teaser
```

The second template in our stylesheet matches anything *except* a valid purchase order. If the root element doesn't have the correct name and namespace URI, the second template stops the processor.

We can use `<xsl:import-schema>` with the schema embedded inside the XSLT stylesheet as well:

```
<?xml version="1.0"?>
<!-- import-schema2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns="http://www.oreilly.com/xslt"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:po="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:import-schema namespace="http://www.oreilly.com/xslt">
    <xs:schema
      targetNamespace="http://www.oreilly.com/xslt"
      xmlns:xs="http://www.w3.org/2001/XMLSchema">

      <xs:element name="purchase-order">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="date"
              minOccurs="1" maxOccurs="1"/>
            <xs:element ref="customer"
              minOccurs="1" maxOccurs="1"/>
            <xs:element ref="items"
              minOccurs="1" maxOccurs="1"/>
          </xs:sequence>
          <xs:attribute name="id" type="xs:string"/>
        </xs:complexType>
      </xs:element>

      <xs:element name="date">
        <xs:complexType>
          <xs:attribute name="year" type="xs:integer"/>
          <xs:attribute name="month" type="xs:integer"/>
          <xs:attribute name="day" type="xs:integer"/>
        </xs:complexType>
      </xs:element>
...
    </xs:schema>
  </xsl:import-schema>
...
  <xsl:template match="schema-element(po:purchase-order)">
    ...
</xsl:stylesheet>
```

It is a fatal error to have an `<xsl:import-schema>` element that has a `select` attribute and content.

Using XML Schemas Without Namespaces

To use an XML Schema without `<xsl:import-schema>`, we have to tie the XML, the schema, and the stylesheet together. We'll start by doing this without namespaces, beginning with the `xsi:noNamespaceSchemaLocation` attribute in our XML document:

```
<?xml version="1.0"?>
<!-- parts-list-schema-no-ns.xml -->
<parts-list
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="parts-list-no-ns.xsd">
  <component component-id="C28392-33-TT">
    <name>Turnip Twaddler</name>
    <partref refid="P81952-26-PK"/>
    <partref refid="P86679-52-SP"/>
    ...
```

The schema doesn't specify a default namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- parts-list-no-ns.xsd -->
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="parts-list">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="component" minOccurs="1" maxOccurs="unbounded"/>
        ...
```

The only prefix we used here is the one for XML Schema itself. The XML file and the schema are now in sync and namespace-free, so our stylesheet is straightforward:

```
<?xml version="1.0"?>
<!-- id-schema-no-ns.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>
  ...
    <xsl:for-each select="/parts-list/part">
      <xsl:text>&#xA;  </xsl:text>
      <xsl:value-of select="name"/>
      <xsl:text> (part #</xsl:text>
      <xsl:value-of select="@part-id"/>
      <xsl:text>) is used in these products:&#xA;    </xsl:text>
      <xsl:for-each
        select="/parts-list/component
                [partref/@refid=current()/@part-id]">
        <xsl:value-of select="name"/>
        ...
```

Because we're not using namespaces, we don't have to qualify anything in our XPath expressions. Next we'll look at using a schema with namespaces.

Using XML Schemas with Namespaces

If we're using an XML Schema with a namespace, we have to make sure the three files (*.xml*, *.xsd*, and *.xsl*) are in sync. It's slightly more complicated, as you'd expect. First of all, the XML file needs to have a default namespace:

```
<?xml version="1.0"?>
<!-- parts-list-schema-ns.xml -->
<parts-list xmlns="http://www.oreilly.com/xslt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oreilly.com/xslt
                      parts-list-schema-ns.xsd">
  <component component-id="C28392-33-TT">
    <name>Turnip Twaddler</name>
    <partref refid="P81952-26-PK"/>
    ...
```

Notice that we've defined the default namespace (`xmlns=`) as `http://www.oreilly.com/xslt`. The `xsi:schemaLocation` attribute has two parts, separated by whitespace: the namespace URI and the URI of the schema itself. The schema file uses the same default namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- parts-list-schema-ns.xsd -->
<xs:schema
  xmlns="http://www.oreilly.com/xslt"
  targetNamespace="http://www.oreilly.com/xslt"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="parts-list">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="component" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element ref="part" minOccurs="1" maxOccurs="unbounded"/>
        ...
```

We also use the XML Schema attribute `targetNamespace`; it has the same value as the default namespace. Finally, we need to use the namespace in the XSLT file. We define `http://www.oreilly.com/xslt` as the default namespace and define a prefix for that namespace as well:

```
<?xml version="1.0"?>
<!-- id-schema-ns.xsl -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.oreilly.com/xslt"
  xmlns:pl="http://www.oreilly.com/xslt">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Here is a test of the id() </xsl:text>
    <xsl:text>function in reverse:&#xA;</xsl:text>
```

```
    <xsl:for-each select="/pl:parts-list/pl:part">
      <xsl:text>&#xA;  </xsl:text>
      <xsl:value-of select="pl:name"/>
      <xsl:text> (part #</xsl:text>
      <xsl:value-of select="@part-id"/>
      <xsl:text>) is used in these products:&#xA;    </xsl:text>
      <xsl:for-each
        select="/pl:parts-list/pl:component
                [pl:partref/@refid=current()/@part-id]">
        <xsl:value-of select="pl:name"/>
```

Because we've defined namespaces in our schema, we have to create a namespace prefix for that namespace and use it whenever we refer to elements in the XML document. If we change the stylesheet to look for `<parts-list>` elements instead of `<pl:parts-list>` elements, the stylesheet won't work. It's more difficult to use a schema with namespaces, but you don't always have a choice.

To address this problem, XSLT 2.0 adds the `xpath-default-namespace` attribute. In our previous stylesheet, we defined a namespace and a namespace prefix, then used the prefix in all of our XPath statements. With `xpath-default-namespace`, we don't have to do that:

```
<?xml version="1.0"?>
<!-- id-schema-ns2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.oreilly.com/xslt"
  xpath-default-namespace="http://www.oreilly.com/xslt">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Here is a test of the id() </xsl:text>
    <xsl:text>function in reverse:&#xA;</xsl:text>

    <xsl:for-each select="/parts-list/part">
      <xsl:text>&#xA;  </xsl:text>
      <xsl:value-of select="name"/>
      <xsl:text> (part #</xsl:text>
      <xsl:value-of select="@part-id"/>
      <xsl:text>) is used in these products:&#xA;    </xsl:text>
      <xsl:for-each
        select="/parts-list/component
                [partref/@refid=current()/@part-id]">
        <xsl:value-of select="name"/>
        <xsl:if test="position() != last()">
          <xsl:text>&#xA;    </xsl:text>
        </xsl:if>
      </xsl:for-each>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

By adding `xpath-default-namespace` to our stylesheet, our XPath expressions are much simpler.

Be aware that you can add this attribute to any element in the XSLT namespace. For example, we could remove `xpath-default-namespace` from the `<xsl:stylesheet>` element and write our stylesheet like this:

```
<?xml version="1.0"?>
<!-- id-schema-ns3.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.oreilly.com/xslt">
...
    <xsl:for-each select="/parts-list/part"
      xpath-default-namespace="http://www.oreilly.com/xslt">
      <xsl:text>&#xA;  </xsl:text>
      <xsl:value-of select="name"/>
...
</xsl:stylesheet>
```

This gives us the same results as before. If you're using data from different namespaces (maybe you're reading more than one XML document as input), redefining the default XPath namespace as needed makes your XPath expressions much simpler. Most of the time, though, you'll just use it on `<xsl:stylesheet>`.

APPENDIX E

[2.0] Regular Expressions

The regular expression language used by XSLT 2.0, XPath 2.0, and XQuery 1.0 is a superset of the XML Schema regular expression language. Regular expressions are used in the XSLT 2.0 [2.0] `<xsl:analyze-string>` element and the XPath 2.0 and XQuery 1.0 functions [2.0] `matches( )`, [2.0] `replace( )`, and [2.0] `tokenize( )`.

This appendix defines the syntax and capabilities of regular expressions in XSLT 2.0, XPath 2.0, and XQuery 1.0. The details of regular expressions are defined in XQuery 1.0 and XPath 2.0 Functions and Operators section 7.6, "String Functions that Use Pattern Matching."

[XPath] The Functions and Operators spec extends the XML Schema regular expression in several ways. Features unique to the XPath 2.0 and XQuery 1.0 regular expression language are indicated with the text *[XPath]*.

Simple Expressions

The simplest regular expression is just a string of text. The regular expression `abc` matches the string `abc`. We're merely searching for a series of characters inside some data. For this case, `contains('abc', 'abc')` does the same thing.

Regular expressions exist to do far more than search for a literal string; they help us find data that matches a pattern. We can use character sets to specify groups of characters. The regular expression `[abc]d` matches two characters in which the first character is `a`, `b`, or `c`, followed by the character `d`.

We can also negate a character set, asking for all the characters *not* in the character set. To do this, add a caret (^) to the start of the character set. The regular expression `[^abc]d` matches two characters in which the first character is anything except `a`, `b`, or `c`, followed by `d`.

Ranges give us a shorthand way of defining character sets. The character set `[0-9]` specifies the digits 0 through 9, while the character set `[a-zA-Z]` specifies all of the

unaccented letters used in Western European languages. A range can be negated just like any other character set; `[^0-9]` specifies anything except the digits.

Ranges can also be subtracted from each other. The range `[A-Z-[IOQ]]` matches any unaccented uppercase letter except `I`, `O`, or `Q`.

Subexpressions

It's often useful to split a regular expression into subexpressions. A subexpression is surrounded by parentheses, and can be modified by a quantifier to define how often (or if) an expression can occur. For example, this regular expression matches a phone number in the format `999-999-9999`:

```
([0-9]{3})-([0-9]{3})-([0-9]{4})
```

The regular expression has three subexpressions, each of which matches a group of digits. XSLT 2.0 provides the `regex-group( )` function to retrieve the part of the analyzed string that matches a particular subexpression. Here's an example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- subexpressions1.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:analyze-string select="'Call me at 919-555-1212, please.'"
      regex="([0-9]{{3}})-(\p{{Nd}}{{3}})-([0-9]{{4}})">
      <xsl:matching-substring>
        <xsl:text>The matching substring is '</xsl:text>

        <!-- <xsl:value-of select="."/> does the same thing here -->
        <xsl:value-of select="regex-group(0)"/>
        <xsl:text>'&#xA;</xsl:text>
        <xsl:text>The formatted string is '</xsl:text>
        <xsl:value-of
          select="('(', regex-group(1), ') ', regex-group(2),
                   '-', regex-group(3), '''')"
          separator=""/>
      </xsl:matching-substring>
    </xsl:analyze-string>
  </xsl:template>

</xsl:stylesheet>
```

(The curly braces in this example are doubled so that the XSLT processor knows this is not an attribute value template.) Notice that the regular expression contains two hyphens that are not part of any subexpression. The function `regex-group(0)` returns the entire portion of the string that matches. That means everything from the first character that matches the regular expression to the last character that matches the regular expression. In our test string here, `regex-group(0)` returns `919-555-1212`, but

not any of the other characters in the string. (Asking for the context item does the same thing.) That includes the two hyphens that aren't in any of the subexpressions. Here are the results of the stylesheet:

```
The matching substring is '919-555-1212'
The formatted string is '(919) 555-1212'
```

Calling `regex-group( )` with a negative number or with a number larger than the number of subexpressions returns an empty string.

The `replace( )` function provides a mechanism similar to `regex-group( )`. Within the replacement string, the dollar sign (`$`) references the matches to different subexpressions. As you would expect, `$1` returns the match for the first subexpression, `$2` returns the match for the second subexpression, and `$0` returns the entire matching string. Here's the `replace( )` version of our previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- subexpressions2.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>The string '</xsl:text>
    <xsl:value-of
      select="replace('Call me at 919-555-1212, please',
              '([0-9]{3})-([0-9]{3})-([0-9]{4})',
              '$0')"/>
    <xsl:text>' contains a match!&#xA;</xsl:text>
    <xsl:text>The version with all the replacements: &#xA;  </xsl:text>
    <xsl:value-of
      select="replace('Call me at 919-555-1212, please',
              '([0-9]{3})-([0-9]{3})-([0-9]{4})',
              '($1) $2-$3')"/>
  </xsl:template>

</xsl:stylesheet>
```

The first call to the `replace( )` function simply uses the `$0` operator. This string is identical to the original string. The `replace( )` function replaces only the part of the string that matches, so replacing the part that matches with the part that matches doesn't change anything. The second call to `replace( )` formats the phone number as we want. We put the area code (`$1`) inside parentheses, followed by a space, the exchange (`$2`), a dash, and the last four digits (`$3`). The results look like this:

```
The string 'Call me at 919-555-1212, please' contains a match!
The version with all the replacements:
  Call me at (919) 555-1212, please
```

There is one important difference between `regex-group()` and `replace()`. Calling `regex-group()` with a group number that does not exist returns an empty string. With the dollar sign, the XSLT 2.0 engine interprets the rightmost digits as literal characters, continuing this process until only a single digit is left. If that single digit is greater than the number of subexpressions, that digit is replaced with an empty string. To quote an example from the spec, if there are five subexpressions, `$23` returns the value of the second subexpression followed by the number `3`. Using `$63` with the same regular expression simply returns the value `3`.

Quantifiers

A quantifier specifies how many times (or if) something occurs. There are three quantifier characters and a syntax for specifying quantities more explicitly. For example, `(a|b)?c` matches `ac`, `bc`, and `c`, because the question mark indicates zero or one instances of a pattern. Here are all the quantifiers and their syntax:

`?`
: Zero or one of a pattern.

`*`
: Zero or more of a pattern.

`+`
: One or more of a pattern.

`{x}`
: The pattern must occur exactly `x` times. The pattern `(a|b){3}` matches `aaa` and `abb`, but not `ab`.

`{x,}`
: The pattern must occur at least `x` times. The pattern `(a|b){3,}` matches `aaa` and `aaaaaaaaa`, but not `aa`.

`{x,y}`
: The pattern must occur at least `x` times, but no more than `y` times. The pattern `(a|b){2,5}` matches `aa` and `aaa`, but not `aaaaaa`.

[XPath] Reluctant Quantifiers

By default, a regular expression or subexpression matches the longest possible string. The quantifiers `+`, `*`, and `?` match as many characters as possible. Adding a question mark to the end of a quantifier causes the expression or subexpression to match the *shortest* possible string. The quantifiers are modified as follows:

`a+?`
: Matches `a` once

`a??`
: Matches `a`, either once or not at all (works only in the `matches( )` function; more on this later in this section)

`a*?`
: Matches `a`, zero times or once (works only in the `matches( )` function; more on this later in this section)

`a{x}?`
: Matches `a`, exactly `x` times (in this case the reluctant qualifier doesn't change anything)

`a{x,}?`
: Matches `a` exactly `x` times (the comma is irrelevant—a reluctant quantifier matches only the minimum length)

`a{x,y}?`
: Matches `a` exactly `x` times (the second number, `y`, is irrelevant—a reluctant quantifier matches only the minimum length)

As an example, we'll use the `replace( )` function. We'll put square brackets around each match in the original string. The normal quantifier and the reluctant quantifier work differently, as we'll see. Here's the stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- reluctant.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Original string: 'Call me at 19195551212'&#xA;</xsl:text>
    <xsl:text>  replace($x, '([0-9]+)', '[$1]'):&#xA;    </xsl:text>
    <xsl:value-of
      select="replace('Call me at 19195551212',
              '([0-9]+)',
              '[$1]')"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:text>  replace($x, '(([0-9])+?)', '[$1]'):&#xA;    </xsl:text>
    <xsl:value-of
      select="replace('Call me at 19195551212',
              '(([0-9])+?)',
              '[$1]')"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:text>  replace($x, '(([0-9]){2,4}?)', '[$1]'):&#xA;    </xsl:text>
    <xsl:value-of
      select="replace('Call me at 19195551212',
              '(([0-9]){2,4}?)',
              '[$1]')"/>
  </xsl:template>

</xsl:stylesheet>
```

Here are the results:

```
Original string: 'Call me at 19195551212'
  replace($x, '([0-9]+)', '[$1]'):
    Call me at [19195551212]
  replace($x, '(([0-9])+?)', '[$1]'):
    Call me at [1][9][1][9][5][5][5][1][2][1][2]
  replace($x, '(([0-9]){2,4}?)', '[$1]'):
    Call me at [19][19][55][51][21]2
```

With the normal quantifier (`([0-9]+)`), the entire string of digits is matched. The single match appears between square brackets. In the second example, the reluctant qualifier (`(([0-9])+?)`) matches each digit separately, so there are square brackets around each digit. The final example reluctantly matches two to four digits, which means each match is two digits long. Notice that the last number in the string isn't part of a match at all.

With the exception of the `matches( )` function, it is illegal to have an expression that matches a zero-length string. The reluctant qualifiers `a??`, `a*?`, `a{0}?`, `a{0,}?`, and `a{0,5}?` all match nothing at all. If you use a regular expression that matches a zero-length string anywhere except the `matches( )` function, the XSLT processor throws a static error.

Processing Modes

There are four flags that change how regular expressions are evaluated:

`s`

Regular expressions are evaluated in what the specs refer to as "dot-all" mode. When this flag is used, the dot operator (`.`) matches any character. Under normal processing (without the `s` flag), the dot operator matches any character *except* the newline character (`#xA`). This flag is useful when you want to match strings that might include a newline character.

Perl and other languages refer to this as "single-line" mode; that's why the abbreviation for "dot-all" mode is `s`.

`m`

Regular expressions are evaulated in multiline mode. By default, the metacharacter (`^`) matches the start of the entire string, while `$` matches the end of the entire string. In multiline mode, `^` matches the start of any line within the string, and `$` matches the end of any line within the string.

`i`

Regular expressions are evaluated in case-insensitive mode. The regular expression `"a"` matches both `"a"` and `"A"`.

Note that Unicode issues can complicate this greatly. For example, the XQuery 1.0 and XPath 2.0 Functions and Operators spec gives the example of the Unicode sign for degrees Kelvin (`K`), which is the letter `"K"`. The combination of `regex="k"` and `flags="i"` matches the Kelvin sign as well as the letters `"k"` (`k`) and `"K"` (`K`).

Other Unicode characters don't convert to letters. For example, the Unicode symbol for the Roman numeral `I` (`Ⅰ`) looks like the letter `I`, but does not convert to one.

`x`

All whitespace characters (`#x9`, `#xA`, `#xD` and `#x20`) are removed from the regular expression before any comparison is done. In other words, with the `x` flag, the regular expressions `"John Smith"` and `"JohnSmith"` are the same. This flag is useful when you want to break a long regular expression into multiple lines to make it easier to read.

The flags can be combined in any order. The attributes `flags="xis"` and `flags="six"` work exactly the same way.

[XPath] Anchors

In XML Schema, regular expressions are anchored; in other words, the regular expression is assumed to start at the beginning of the text and end at the end of the text. That means the expression `abc` matches only the three-character string `abc`. If there are any extra characters before or after the letters `abc`, XML Schema does not consider the string a match.

The regular expression language in XPath 2.0 doesn't work that way. When we use any of the regular expression functions or elements, a string matches if it contains the regular expression *anywhere* inside it. In other words, using XPath's regular expression language, both `abc` and `I know my abc's` match the expression `abc`. XPath provides the traditional anchor operators used in other regular expression languages. The caret (`^`) matches the start of the string, while the dollar sign (`$`) matches the end of the string. If multiline mode is on (`-m`), the caret matches the start of the string and any character immediately following a newline. Similarly, in multiline mode, the dollar sign matches the end of the string and any character immediately before a newline.

Here's an example that illustrates how the anchors work:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- anchors.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>matches('abcdefghij', 'cde'): </xsl:text>
```

```
    <xsl:value-of
      select="if (matches('abcdefghij', 'cde'))
              then 'It''s a match!'
              else 'No match'"/>
    <xsl:text>&#xA;matches('abcdefghij', 'cde$'): </xsl:text>
    <xsl:value-of
      select="if (matches('abcdefghij', 'cde$'))
              then 'It''s a match!'
              else 'No match'"/>
    <xsl:text>&#xA;matches('abcdefghij', 'hij$'): </xsl:text>
    <xsl:value-of
      select="if (matches('abcdefghij', 'hij$'))
              then 'It''s a match!'
              else 'No match'"/>
    <xsl:text>&#xA;matches('ab&amp;#xA;cdefghij', '^cde', 'm'): </xsl:text>
    <xsl:value-of
      select="if (matches('ab&#xA;cdefghij', '^cde', 'm'))
              then 'It''s a match!'
              else 'No match'"/>
  </xsl:template>

</xsl:stylesheet>
```

The first example doesn't use an anchor at all, so `matches( )` returns `true`. The second test looks for the characters `cde` at the end of the string, which means `matches( )` is `false`. In the third example, `hij` is at the end of the string, so we have a match. In the final example, we insert a newline character (`
`) inside the string and use multiline mode. Looking for `cde` at the start of a line succeeds. Here are the results of the stylesheet:

```
matches('abcdefghij', 'cde'): It's a match!
matches('abcdefghij', 'cde$'): No match
matches('abcdefghij', 'hij$'): It's a match!
matches('ab&#xA;cdefghij', '^cde', 'm'): It's a match!
```

If you want your regular expressions to work the way they do in XML Schema, simply surround your regular expressions with `^` and `$`. The expression `^abc$` matches the three-character string `abc`, while the expression `abc` matches any string of any length that contains the three characters `abc` at least once.

Back-references

A back-reference allows you to refer to a previously matched subexpression *within the regular expression itself*. As an example, say you want to find words that begin and end with the same two letters. The expression `([a-z]{2})(.*)\1` does the trick. The `\1` represents whatever two characters matched the first subexpression. The back-reference `\2` represents the match for the second subexpression, `\3` represents the match for the third subexpression, and so on. (Notice that although the references go backwards, counting subexpressions goes from left to right.)

Here's a sample stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- back-reference.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>Using back-references to find words that begin &#xA;</xsl:text>
    <xsl:text>  and end with the same two letters:&#xA;&#xA;</xsl:text>
    <xsl:text>  replace($x, '([a-z]{2})(.*)\1', '$1--$2--$1')&#xA;</xsl:text>
    <xsl:text>    edited:    </xsl:text>
    <xsl:value-of
      select="replace('edited', '([a-z]{2})(.*)\1', '$1--$2--$1')"/>
    <xsl:text>&#xA;    editor:    </xsl:text>
    <xsl:value-of
      select="replace('editor', '([a-z]{2})(.*)\1', '$1--$2--$1')"/>
    <xsl:text>&#xA;    educated:  </xsl:text>
    <xsl:value-of
      select="replace('educated', '([a-z]{2})(.*)\1', '$1--$2--$1')"/>
    <xsl:text>&#xA;    orator:    </xsl:text>
    <xsl:value-of
      select="replace('orator', '([a-z]{2})(.*)\1', '$1--$2--$1')"/>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet generates these results:

```
Using back-references to find words that begin
  and end with the same two letters:

  replace($x, '([a-z]{2})(.*)\1', '$1--$2--$1')
    edited:    ed--it--ed
    editor:    editor
    educated:  ed--ucat--ed
    orator:    or--at--or
```

For the words `edited`, `educated`, and `orator`, the regular expression matches. The `replace( )` function writes out the two letters at the start and end of the word, with whatever was in between them offset by double hyphens. The word `editor` doesn't match the regular expression, so it isn't changed.

Metacharacters

Within a regular expression, most characters represent themselves. For example, the regular expression `A` represents a capital A. There are, of course, special characters that are processed differently:

.

By default, this matches any character except the newline character. In dot-all mode, this matches the newline character as well.

`^`

By default, this represents the beginning of the string literal. In multiline mode, this represents the beginning of a line within the string.

[XPath] Use of the caret to indicate that the beginning of a string or line is an addition to the regular expression syntax defined by XML Schema.

The caret can also be used inside a character class expression to indicate the negation of that character set. For example, `[a-f]` represents the letters `a` through `f`, while `[^a-f]` represents every character *except* the letters `a` through `f`.

`$`

[XPath] By default, this represents the end of the string literal. In multiline mode, this represents the end of a line within the string.

`\`

Escapes the following character.

`|`

The union operator. The expression `A|B` matches both `A` and `B`. It does not match `AB`.

`?`

Zero or one of a pattern. For example, `A[A-Z]?Z` matches `AZ` and `ABZ`, but not `ABCZ`.

`*`

Zero or more of a pattern. For example, `A[A-Z]*` matches any string of uppercase basic Latin characters that starts with A and is followed by zero or more uppercase Latin characters. The strings `A`, `ABC`, `AA`, and `AREALLYLONGSTRING` all match this expression.

`+`

One or more of a pattern. For example, `A[A-Z]+` matches any string of uppercase basic Latin characters that starts with A and is followed by one or more uppercase characters. The strings `AA`, `AD`, and `ADD` match this expression, but `A` does not.

`(`

Starts a subexpression. For example, in the expression `A(BCD)+`, the parentheses define the subexpression `BCD`, which can appear once or not at all. This expression matches `A` and `ABCD`.

`)`

Ends a subexpression.

`[`

Begins a character class expression. For example, the expression `[a-f]` refers to the letters `a` through `f`.

`-`

Within square brackets, separates the upper and lower bounds of a range. For example, `[a-f]` represents the letters `a` through `f`. The expressions `[a-f]` and `a|b|c|d|e|f` are equivalent.

]
: Ends a character class expression.

{
: Starts a named category, block, or quantifier. For example, `\p{IsThai}` specifies the set of all Thai characters (`&#x0E00` to `&#x0E7F`), while `\p{Lu}` specifies all uppercase letters.

  Quantifiers can have three forms: `{x}`, `{x,}`, and `{x,y}`, where `x` and `y` are integers and `y` is greater than `x`. The quantifier `{2}` means something must occur exactly two times, `{2,}` means at least two times, and `{2,5}` means two to five times. The expression `(AB|AC|AD){2,5}` matches `ACAB` and `ACADACADAB`, but not `AB` or `ABABABABABAB`.

}
: Ends a named category, block, or quantifier.

Because curly braces ({ and }) identify attribute value templates, you must use double curly braces for a regular expression inside an attribute. For example, if you want to look for a five-digit number with the `<xsl:analyze-string>` element, you would code this:

```
<xsl:analyze-string select="." regex="[0-9]{{5}}" ...>
```

If you don't double the curly braces inside an attribute, the regular expression `[0-9]{5}` (single curly braces) is interpreted as a single digit followed by the number `5`.

On the other hand, when using a regular expression in a function call, you *don't* use double curly braces. To replace a five-digit number with dashes, you would code this:

```
<xsl:value-of select="replace(., [0-9]{5}, '-----')"/>
```

Single-Character Escapes

The regular expression language provides escapes for some special characters. They are:

\n
: The newline character (`
`)

\r
: The return character (`
`)

\t
: The tab character (`	`)

\\
: The backslash character (`\`)

\|
: The vertical bar character (`|`)

`\.`
: A period (`.`)

`\-`
: The hyphen or minus character (`-`)

`\^`
: The circumflex accent (`^`)

`\?`
: A question mark (`?`)

`\*`
: An asterisk (`*`)

`\+`
: The plus sign (`+`)

`\{`
: A left curly bracket (`{`)

`\}`
: A right curly bracket (`}`)

`\(`
: A left parenthesis (`(`)

`\)`
: A right parenthesis (`)`)

`\[`
: A left square bracket (`[`)

`\]`
: A right square bracket (`]`)

Multiple-Character Escapes

There are also escapes that represent multiple characters. They are:

`\s`
: Any whitespace character

`\S`
: Anything other than a whitespace character

`\d`
: Any digit

`\D`
: Anything other than a digit

`\w`
: Any word character. Word characters are all characters except the punctuation, separator and other characters (the character groups defined with P, Z and C).

`\W`
: Anything other than a word character.

`\i`
: Any character that can be the first letter of an XML name. That includes the underscore (`_`), the colon (`:`), and characters of all the world's languages defined as letters in the XML specification.

`\I`
: Anything other than a character that can be the first letter of an XML name.

`\c`
: Any character that can be used in an XML name. That includes all of the characters that can appear as the first character (`\i`), plus the period (`.`), hyphen (`-`), and the characters defined as digits, combining characters and extenders in the XML specification.

`\C`
: Anything other than a character that can be used in an XML name.

`\p`
: A character in the following named character group. For example, `\p{Nd}` matches a numeric digit, while `\p{IsThai}` matches a Thai character.

`\P`
: Anything other than a character in the following named character group. For example, `\P{Nd}` matches anything except a numeric digit, while `\P{IsThai}` matches anything except a Thai character.

Character Groups

For convenience, we can use character groups to stand for groups of characters. Specifying `L` is simpler than specifying all letters, particularly if you have to allow for the many thousands of characters defined in the Unicode standard. The character groups are used with the `\p` and `\P` operators; `\p{L}` means all letters, while `\P{L}` means everything *except* letters. The character groups and their names are defined in the Unicode Character Database, available online here: *http://www.unicode.org/Public/UNIDATA/UCD.html*.

The following lists are the character groups.

Letters

`L`
: All letters.

`Lu`
: All uppercase letters.

`Ll`
: All lowercase letters.

`Lt`
: Letters in titlecase, such as `&#1C5;`, which is a single character that combines a capital letter D with a small z that has a caron.

`Lm`
: Modifiers (characters such as `ª`, which is the feminine ordinal indicator [ª]).

`Lo`
: Other letters, including ideographs. The Hebrew letter alef (`א`) is an example.

Marks

`M`
: All marks.

`Mn`
: Nonspacing marks (characters such as `́`, which is the Unicode combining acute accent).

`Mc`
: Spacing combining marks.

`Me`
: Enclosing marks.

Numbers

`N`
: All numbers, including alternate forms such as circled (Unicode character `①`, which is the circled digit one) and parenthesized (Unicode character `প`, the parenthesized digit one).

`Nd`
: Decimal digits (as opposed to `Ⅶ`, which is the Unicode character for the Roman numeral seven).

`Nl`
: Numbers as letters (as opposed to digits; the aforementioned Unicode character `Ⅶ` character applies here).

`No`
: Other numeric characters, such as `½`, which is the single character for the fraction ½.

Punctuation

`P`
: All punctuation characters.

`Pc`
: Connector characters, such as `_`, which is the underscore character.

`Pd`
: Dashes, such as the hyphen-minus character, `-`.

`Ps`
: Opening (start) characters, such as a left curly bracket ({).

`Pe`
: Closing characters, such as a right curly bracket (}).

`Pi`
: Initial quote marks. Be aware that in some languages, an initial quote mark might actually be in the set of closing characters (`Pe`). As an example, quotations in Swedish begin and end with a double closing quotation mark (`ȝ`). In that case, the initial quote mark would not be found by searching for `Pi`.

`Pf`
: Final quote marks. As with initial quote marks, a character used as a final quote mark in a particular language can be from the set of opening or closing characters.

`Po`
: Other punctuation marks.

Separators

`Z`
: All separators

`Zs`
: Space characters, such as an em space (Unicode character ` `) or an en space (Unicode character ` `)

`Zl`
: The line separator character (Unicode character ` `)

`Zp`
: The paragraph separator character (Unicode character ` `)

Symbols

`S`
: All symbols

`Sm`
: Math symbols

`Sc`
: Currency symbols

`Sk`
: Modifiers

`So`
: Other symbols

Everything Else

`C`
: All others

`Cc`
: Control characters

`Cf`
: Formatting characters

`Co`
: Private use (reserved by Unicode)

`Cn`
: Not assigned

The Unicode standard also defines a class of "surrogate" characters, `Cs`. These characters don't apply to XML documents; by the time a document is parsed, the surrogate characters no longer exist. For this reason, `Cs` is not a valid character group.

Block Escapes

A block escape is a simple way to refer to a range of characters that have some property in common. Each block escape has a name; to use a block name, prepend `Is` to it. Block escapes are used with the `\p` and `\P` operators. For example, the expression `\p{IsThai}` refers to the Thai characters (`&#x0E00` – `&#x0E7F`). The expression `\P{IsThai}` refers to everything except Thai characters. The block names are listed here in the format defined in the XML Schema spec.

Table E-1 shows the complete list of block escapes. This table was generated from version 5.0.0 of the file *blocks.txt*. The list of block escape names is part of the Unicode Character Database; see *http://www.unicode.org/* for the latest version of the Unicode standard.

Table E-1. Block escape names

Block name	Starting character	Ending character
BasicLatin	&#x0000	&#x007F
Latin-1Supplement	&#x0080	&#x00FF
LatinExtended-A	&#x0100	&#x017F
LatinExtended-B	&#x0180	&#x024F
IPAExtensions	&#x0250	&#x02AF
SpacingModifierLetters	&#x02B0	&#x02FF
CombiningDiacriticalMarks	&#x0300	&#x036F
GreekandCoptic	&#x0370	&#x03FF
Cyrillic	&#x0400	&#x04FF
CyrillicSupplement	&#x0500	&#x052F
Armenian	&#x0530	&#x058F
Hebrew	&#x0590	&#x05FF
Arabic	&#x0600	&#x06FF
Syriac	&#x0700	&#x074F
ArabicSupplement	&#x0750	&#x077F
Thaana	&#x0780	&#x07BF
NKo	&#x07C0	&#x07FF
Devanagari	&#x0900	&#x097F
Bengali	&#x0980	&#x09FF
Gurmukhi	&#x0A00	&#x0A7F
Gujarati	&#x0A80	&#x0AFF
Oriya	&#x0B00	&#x0B7F
Tamil	&#x0B80	&#x0BFF
Telugu	&#x0C00	&#x0C7F
Kannada	&#x0C80	&#x0CFF
Malayalam	&#x0D00	&#x0D7F
Sinhala	&#x0D80	&#x0DFF
Thai	&#x0E00	&#x0E7F
Lao	&#x0E80	&#x0EFF
Tibetan	&#x0F00	&#x0FFF
Myanmar	&#x1000	&#x109F
Georgian	&#x10A0	&#x10FF
HangulJamo	&#x1100	&#x11FF
Ethiopic	&#x1200	&#x137F

Block name	Starting character	Ending character
EthiopicSupplement	&#x1380	&#x139F
Cherokee	&#x13A0	&#x13FF
UnifiedCanadianAboriginalSyllabics	&#x1400	&#x167F
Ogham	&#x1680	&#x169F
Runic	&#x16A0	&#x16FF
Tagalog	&#x1700	&#x171F
Hanunoo	&#x1720	&#x173F
Buhid	&#x1740	&#x175F
Tagbanwa	&#x1760	&#x177F
Khmer	&#x1780	&#x17FF
Mongolian	&#x1800	&#x18AF
Limbu	&#x1900	&#x194F
TaiLe	&#x1950	&#x197F
NewTaiLue	&#x1980	&#x19DF
KhmerSymbols	&#x19E0	&#x19FF
Buginese	&#x1A00	&#x1A1F
Balinese	&#x1B00	&#x1B7F
PhoneticExtensions	&#x1D00	&#x1D7F
PhoneticExtensionsSupplement	&#x1D80	&#x1DBF
CombiningDiacriticalMarksSupplement	&#x1DC0	&#x1DFF
LatinExtendedAdditional	&#x1E00	&#x1EFF
GreekExtended	&#x1F00	&#x1FFF
GeneralPunctuation		&#x206F
SuperscriptsandSubscripts	&#x2070	&#x209F
CurrencySymbols	&#x20A0	&#x20CF
CombiningDiacriticalMarksforSymbols	&#x20D0	&#x20FF
LetterlikeSymbols	&#x2100	&#x214F
NumberForms	&#x2150	&#x218F
Arrows	&#x2190	&#x21FF
MathematicalOperators	&#x2200	&#x22FF
MiscellaneousTechnical	&#x2300	&#x23FF
ControlPictures	&#x2400	&#x243F
OpticalCharacterRecognition	&#x2440	&#x245F
EnclosedAlphanumerics	&#x2460	&#x24FF
BoxDrawing	&#x2500	&#x257F

Block name	Starting character	Ending character
BlockElements	&#x2580	&#x259F
GeometricShapes	&#x25A0	&#x25FF
MiscellaneousSymbols	&#x2600	&#x26FF
Dingbats	&#x2700	&#x27BF
MiscellaneousMathematicalSymbols-A	&#x27C0	&#x27EF
SupplementalArrows-A	&#x27F0	&#x27FF
BraillePatterns	&#x2800	&#x28FF
SupplementalArrows-B	&#x2900	&#x297F
MiscellaneousMathematicalSymbols-B	&#x2980	&#x29FF
SupplementalMathematicalOperators	&#x2A00	&#x2AFF
MiscellaneousSymbolsandArrows	&#x2B00	&#x2BFF
Glagolitic	&#x2C00	&#x2C5F
LatinExtended-C	&#x2C60	&#x2C7F
Coptic	&#x2C80	&#x2CFF
GeorgianSupplement	&#x2D00	&#x2D2F
Tifinagh	&#x2D30	&#x2D7F
EthiopicExtended	&#x2D80	&#x2DDF
SupplementalPunctuation	&#x2E00	&#x2E7F
CJKRadicalsSupplement	&#x2E80	&#x2EFF
KangxiRadicals	&#x2F00	&#x2FDF
IdeographicDescriptionCharacters	&#x2FF0	&#x2FFF
CJKSymbolsandPunctuation		&#x303F
Hiragana	&#x3040	&#x309F
Katakana	&#x30A0	&#x30FF
Bopomofo	&#x3100	&#x312F
HangulCompatibilityJamo	&#x3130	&#x318F
Kanbun	&#x3190	&#x319F
BopomofoExtended	&#x31A0	&#x31BF
CJKStrokes	&#x31C0	&#x31EF
KatakanaPhoneticExtensions	&#x31F0	&#x31FF
EnclosedCJKLettersandMonths	&#x3200	&#x32FF
CJKCompatibility	&#x3300	&#x33FF
CJKUnifiedIdeographsExtensionA	&#x3400	&#x4DBF
YijingHexagramSymbols	&#x4DC0	&#x4DFF
CJKUnifiedIdeographs	&#x4E00	&#x9FFF

Block name	Starting character	Ending character
YiSyllables	&#xA000	&#xA48F
YiRadicals	&#xA490	&#xA4CF
ModifierToneLetters	&#xA700	&#xA71F
LatinExtended-D	&#xA720	&#xA7FF
SylotiNagri	&#xA800	&#xA82F
Phags-pa	&#xA840	&#xA87F
HangulSyllables	&#xAC00	&#xD7AF
HighSurrogates	&#xD800	&#xDB7F
HighPrivateUseSurrogates	&#xDB80	&#xDBFF
LowSurrogates	&#xDC00	&#xDFFF
PrivateUseArea	&#xE000	&#xF8FF
CJKCompatibilityIdeographs	&#xF900	&#xFAFF
AlphabeticPresentationForms	&#xFB00	&#xFB4F
ArabicPresentationForms-A	&#xFB50	&#xFDFF
VariationSelectors	&#xFE00	&#xFE0F
VerticalForms	&#xFE10	&#xFE1F
CombiningHalfMarks	&#xFE20	&#xFE2F
CJKCompatibilityForms	&#xFE30	&#xFE4F
SmallFormVariants	&#xFE50	&#xFE6F
ArabicPresentationForms-B	&#xFE70	&#xFEFF
HalfwidthandFullwidthForms	&#xFF00	&#xFFEF
Specials	&#xFFF0	&#xFFFF
LinearBSyllabary	&#x10000	&#x1007F
LinearBIdeograms	&#x10080	&#x100FF
AegeanNumbers	&#x10100	&#x1013F
AncientGreekNumbers	&#x10140	&#x1018F
OldItalic	&#x10300	&#x1032F
Gothic	&#x10330	&#x1034F
Ugaritic	&#x10380	&#x1039F
OldPersian	&#x103A0	&#x103DF
Deseret	&#x10400	&#x1044F
Shavian	&#x10450	&#x1047F
Osmanya	&#x10480	&#x104AF
CypriotSyllabary	&#x10800	&#x1083F
Phoenician	&#x10900	&#x1091F

Block name	Starting character	Ending character
Kharoshthi	&#x10A00	&#x10A5F
Cuneiform	&#x12000	&#x123FF
CuneiformNumbersandPunctuation	&#x12400	&#x1247F
ByzantineMusicalSymbols	&#x1D000	&#x1D0FF
MusicalSymbols	&#x1D100	&#x1D1FF
AncientGreekMusicalNotation	&#x1D200	&#x1D24F
TaiXuanJingSymbols	&#x1D300	&#x1D35F
CountingRodNumerals	&#x1D360	&#x1D37F
MathematicalAlphanumericSymbols	&#x1D400	&#x1D7FF
CJKUnifiedIdeographsExtensionB	&#x20000	&#x2A6DF
CJKCompatibilityIdeographsSupplement	&#x2F800	&#x2FA1F
Tags	&#xE0000	&#xE007F
VariationSelectorsSupplement	&#xE0100	&#xE01EF
SupplementaryPrivateUseArea-A	&#xF0000	&#xFFFFF
SupplementaryPrivateUseArea-B	&#x100000	&#x10FFFF

You can find the latest list of block names and the characters they contain at Unicode.org in the file *blocks.txt*.

APPENDIX F
XSLT Formatting Codes

This appendix lists the formatting codes for numbers, dates, and times used in XSLT. These codes are used in the following elements and functions:

- `format-date( )`, `format-time( )` and `format-dateTime( )`, defined in XSLT 2.0 section 16.5, "Formatting Dates and Times."
- The `format` attribute of the `<xsl:number>` element, defined in XSLT 1.0 section 7.7, "Numbering." The capabilities of the `format` attribute were enhanced in XSLT 2.0 section 12, "Numbering."
- The `<xsl:decimal-format>` element, defined in XSLT 1.0 section 12.3, "Number Formatting."
- The `format-number( )` function, defined in XSLT 1.0 and XSLT 2.0.

Most of the formatting codes for numbers were defined in XSLT 1.0. The main additions for number formatting added by XSLT 2.0 are the ordinal and language options. The date and time formatting functions were added in XSLT 2.0. There are three types of formatting codes we'll list here:

- The date and time format codes defined by the ISO 8601 standard for the representation of dates and times. You use these format codes for creating new `xs:date`, `xs:time`, `xs:dateTime`, `xs:duration`, `xs:dayTimeDuration`, and `xs:yearMonthDuration` values.
- The date and time formatting codes defined in XSLT 2.0 section 16.5, "Formatting Dates and Times." Use these codes to format parts of `xs:date`, `xs:time`, `xs:dateTime`, `xs:duration`, `xs:dayTimeDuration`, and `xs:yearMonthDuration` values.
- The numbering formatting codes defined in XSLT 1.0 section 7.7, "Numbering," and section 12.3, "Number Formatting." These codes were extended in XSLT 2.0 section 12, "Numbering," and section 16.4, "Number Formatting."

Formatting Codes for Numbers

The following subsections detail the codes for formatting numbers. These are used by the `<xsl:decimal-format>` and `<xsl:number>` elements and by the `format-number( )` function.

Parts of Numbers

Table F-1 describes the characters that represent parts of a number. They are used by the `format-number( )` function and by the `<xsl:decimal-format>` element.

Table F-1. Number formatting codes

Code	Meaning
`#`	Represents a digit. Trailing or leading zeroes are not displayed. Formatting the number 4.0 with the string `"#.##"` returns the string `"4"`.
`0`	Represents a digit. Unlike the # character, the 0 always displays a zero. Formatting the number 4.1 with the string `"#.00"` returns the string `"4.10"`.
`.`	Represents the decimal point.
`-`	Represents the minus sign.
`,`	Is the grouping separator.
`;`	Separates the positive-number pattern from the negative-number pattern.
`%`	Indicates that a number should be displayed as a percentage. The value is multiplied by 100, and then displayed as a percentage. Formatting the number `.76` with the string `"##%"` returns the string `"76%"`.
`\u2030`	Is the Unicode character for the per-thousand (per-mille) sign. The value will be multiplied by 1000, and then displayed as a per mille. Formatting the number `.768` with the string `"###\u2030"` returns the string `"768‰"`.

Parts of Decimal Formats

The `<xsl:decimal-format>` element lets us define named formats for decimal numbers. Each named format is a group of settings for the properties discussed in Table F-2.

Table F-2. Parts of decimal formats

Part name	Meaning	Default value
`name`	Gives a name to this format.	`N/A`
`decimal-separator`	Defines the character (usually either a period or comma) used as the decimal point. This character is used both in the format string and in the output.	`.` (period)
`grouping-separator`	Defines the character (usually either a period or comma) used as the thousands separator. This character is used both in the format string and in the output.	`,` (comma)
`infinity`	Defines the string used to represent infinity. Be aware that XSLT's number facilities support both positive and negative infinity. This string is used only in the output.	`Infinity`

Part name	Meaning	Default value
minus-sign	Defines the character used as the minus sign. This character is used only in the output.	- (hyphen, -)
NaN	Defines the string displayed when the value to be formatted is not a number. This string is used only in the output.	NaN
percent	Defines the character used as the percent sign. This character is used both in the format string and in the output.	% (percent sign)
per-mille	Defines the character used as the per-mille sign. This character is used both in the format string and in the output.	‰ (Unicode per-mille character, ‰).
zero-digit	Defines the character used for the digit zero. This character is used both in the format string and in the output.	0 (zero)
digit	Defines the character used in the format string to stand for a digit.	#
pattern-separator	Defines the character used to separate the positive and negative subpatterns in a pattern. This character is used only in the format string.	; (semicolon)

Formatting Codes for Dates and Times

Parts of Dates and Times

Table F-3 describes the codes defined for the different parts of `xs:date`, `xs:dateTime`, and `xs:time` values.

Table F-3. Parts of dates and times

Specifier	Meaning	Default presentation modifier
Y	Year (absolute value)	1
M	Month in year	1
D	Day in month	1
d	Day in year	1
F	Day of week	n
W	Week in year	1
w	Week in month	1
H	Hour in day (24 hours)	1
h	Hour in half-day (12 hours)	1
P	a.m./p.m. marker	n
m	Minute in hour	01
s	Second in minute	01
f	Fractional seconds	1
Z	Timezone as a time offset from UTC—or if an alphabetic modifier is present, the conventional name of a timezone (such as PST)	1

Specifier	Meaning	Default presentation modifier
z	Timezone as a time offset using GMT—for example, GMT+1	1
C	Calendar: the name or abbreviation of a calendar name	n
E	Era: the name of a baseline for the numbering of years—for example, the reign of a monarch	n

Presentation Modifiers

A presentation modifier is a code that defines how a part of a date or time value should be displayed. For example, FNn displays the day of the week as the capitalized name of the day, using the current language. If your system is set up to use English and the calendar used in the United States, the name of the day of the week (FNn) for 15 August 2007 is Wednesday. The formatting code FWw displays the numeric value Four. Using German with the same codes on the same date displays the values Mittwoch and Drei (Wednesday is the third day of the week in the European calendar). Table F-4 describes the presentation modifiers.

Table F-4. Presentation modifiers

Modifier	Meaning
1	The value as a number with one or more digits: 0, 1, 2, 3, 4, ...
01	The value as a number with two or more digits: 00, 01, 02, ... 09, 10, 11, ... 99, 100, 101, ...
(Other numeric values)	An XSLT processor is free to support other numbering schemes. To quote an example from the spec, a formatting code of &#xOE51; tells the processor to use Thai numbering. See the documentation for your processor to see which numbering schemes it supports.
A	The value as an uppercase letter: A, B, C, ... Z, AA, AB, AC, ...
a	The value as a lowercase letter: a, b, c, ... z, aa, ab, ac, ...
I	The value as an uppercase roman numeral: I, II, III, IV, V, VI, VII, VIII, IX, X, ...
i	The value as an lowercase roman numeral: i, ii, iii, iv, v, vi, vii, viii, ix, x, ...
n	The name of the component in lowercase. Applies only to components that have names, such as the day of the week (F), the month of the year (M), and the time zone (Z).
N	The name of the component in uppercase. Applies only to components that have names, such as the day of the week (F), the month of the year (M), and the time zone (Z).
Nn	The name of the component in title case. Applies only to components that have names, such as the day of the week (F), the month of the year (M), and the time zone (Z).
w	The value as a word in lowercase: one, two, three, four, ...
W	The value as a word in uppercase: ONE, TWO, THREE, FOUR, ...
Ww	The value as a word in title case: One, Two, Three, Four, ...

Calendars

These are the calendar names defined in the XSLT 2.0 specification. In addition to the values listed in Table F-5, an XSLT processor is free to implement support for other calendars. The abbreviation for any calendar not listed in the table must be qualified with a namespace prefix. Check the documentation for your XSLT processor to see which calendars it supports.

Table F-5. Calendars

Abbreviation	Calendar
AD	Anno Domini (Christian Era)
AH	Anno Hegirae (Muhammedan Era)
AME	Mauludi Era (solar years since Mohammed's birth)
AM	Anno Mundi (Jewish Calendar)
AP	Anno Persici
AS	Aji Saka Era (Java)
BE	Buddhist Era
CB	Cooch Behar Era
CE	Common Era
CL	Chinese Lunar Era
CS	Chula Sakarat Era
EE	Ethiopian Era
FE	Fasli Era
ISO	ISO 8601 calendar
JE	Japanese Calendar
KE	Khalsa Era (Sikh calendar)
KY	Kali Yuga
ME	Malabar Era
MS	Monarchic Solar Era
NS	Nepal Samwat Era
OS	Old Style (Julian Calendar)
RS	Rattanakosin (Bangkok) Era
SE	Saka Era
SH	Mohammedan Solar Era (Iran)
SS	Saka Samvat
TE	Tripurabda Era
VE	Vikrama Era
VS	Vikrama Samvat Era

APPENDIX G

XSLT 2.0 Migration Guide

This appendix contains some hits and suggestions for migrating your XSLT 1.0 stylesheets to XSLT 2.0. Most XSLT 1.0 stylesheets will work with an XSLT 2.0 processor, but there are a few errors you need to watch for. We'll also look at some of the new features in XSLT 2.0. If none of the new features in XSLT 2.0 and XPath 2.0 provide any value for you, there's no reason to migrate your stylesheets. (It's very unlikely that nothing in XSLT 2.0 and XPath 2.0 can help you, but still....)

Powerful New Features in XSLT 2.0 and XPath 2.0

We'll discuss some of the most important new features in XSLT 2.0 and XPath 2.0. This isn't a complete list, but it should give you an idea of whether XSLT 2.0 will simplify your stylesheets and applications.

Recursion Isn't Necessary Nearly as Often

In XSLT 1.0 stylesheets, tail recursion was a common technique. String replacement, for example, requires a recursive template that replaces the first instance of a substring, and then reinvokes itself with the remainder of the string. In XSLT 2.0, you can use the `replace( )` function. Recursive templates are much more difficult to write and maintain; if you can replace them with simpler code in an XSLT 2.0 stylesheet, it's worth it. See the section "[2.0] Using the XPath 2.0 replace() Function to Avoid Recursion" in Chapter 5 for a more detailed example. Another good example is in the section "Doing Math Without Recursion" in Chapter 8.

Grouping Is Much, Much Easier

Grouping in XSLT 2.0 is orders of magnitude easier than it was in XSLT 1.0. The `<xsl:for-each-group>` element can make your code as much as 80% smaller, depending on the complexity of your XSLT 1.0 stylesheets. If your existing stylesheets do grouping, the new grouping support in XSLT 2.0 can make them much more elegant. See the section "[2.0] New Grouping Syntax in XSLT 2.0" in Chapter 7 for all the details.

Datatypes and XML Schemas Are Supported

If type checking and validation are important to your application, moving to XSLT 2.0 is a must. A basic XSLT 2.0 processor supports all the basic datatypes of XML Schema, while a schema-aware processor lets you define your own datatypes and use them in your stylesheets. Datatypes and schema support are discussed in Chapter 3.

Regular Expressions Are Supported

Although XSLT 1.0 supports functions such as `contains( )`, `substring-before( )`, and `substring-after( )`, these are no substitute for regular expressions. The regular expression support in XSLT 2.0 is very powerful and flexible. See the discussions of the [2.0] `<xsl:analyze-string>` element and the [2.0] `matches( )`, [2.0] `replace( )`, and [2.0] `tokenize( )` functions for the details on regular expression support. Appendix E has a complete overview of the regular expression syntax used in XSLT 2.0 and XPath 2.0.

Potential Errors

There are several differences between XSLT 1.0 and XSLT 2.0 that can cause fatal errors. If you're going to migrate your 1.0 stylesheets to 2.0, you'll need to handle all of those differences.

Passing Undefined Parameters with <xsl:call-template> Causes an Error

In XSLT 1.0, you could use `<xsl:call-template>` to invoke a template with as many parameters as you wanted. If any of those parameters weren't defined in the template, they were ignored. In XSLT 2.0, using `<xsl:call-template>` with an undefined parameter causes a fatal error.

It's *not* an error to pass undefined parameters with `<xsl:apply-templates>`, `<xsl:apply-imports>`, or `<xsl:next-match>`.

Many times XSLT 1.0 stylesheets passed extra parameters because they might eventually be needed by another template. Every time another template was invoked (via `<xsl:apply-templates>` or `<xsl:call-template>`), the extra parameters were passed along. XSLT 2.0 features much more sophisticated *tunnel parameters*. A tunnel parameter isn't passed to each template, so it makes your stylesheets much cleaner. See the section "Tunnel parameters" in Chapter 5 for more information.

This potential problem is a static error. That means the XSLT 2.0 processor can tell whether you're passing the wrong number of parameters to a template without actually transforming an XML document.

Math Works Differently in Some Cases

XSLT 1.0 featured a simple `number` datatype. Math operations, particularly anything dealing with division by zero, work differently in XSLT 2.0.

To make things more complicated, XSLT 2.0 handles different kinds of numbers differently. Dividing an `xs:integer` or `xs:decimal` by zero is a fatal error, while dividing an `xs:double` or `xs:float` returns `INF` (infinity).

The simplest way to avoid this error is to add `version="1.0"` to the element. That causes the XSLT 2.0 processor to evaluate the expression in XSLT 1.0 mode, so your stylesheets will behave as they always have.

```
<xsl:variable version="1.0" name="ratio"
  select="$orders div $returns"/>
```

A more sophisticated approach is to use the new XPath `if` operator. For example, you could change the code like this:

```
<xsl:variable name="ratio"
  select="if ($returns != 0) then
          $orders div $returns else
          0"/>
```

If the value of `$returns` is not equal to zero, we perform the calculation; otherwise, we return zero.

Division works with six datatypes: `xs:integer`, `xs:decimal`, `xs:float`, `xs:double`, `xs:yearMonthDuration`, and `xs:dayTimeDuration`. Here's a stylesheet that illustrates how division by zero works:

```
<?xml version="1.0"?>
<!-- divide-by-zero.xsl -->
<xsl:stylesheet version="2.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#xA;Division by zero in XSLT 2.0:&#xA;&#xA;</xsl:text>

    <xsl:text>&lt;xsl:value-of select="1 div 0"/&gt;&#xA;</xsl:text>
    <xsl:text>  [fatal error]&#xA;&#xA;</xsl:text>
    <xsl:text>&lt;xsl:value-of select="1.0 div 0.0"/&gt;&#xA;</xsl:text>
    <xsl:text>  [fatal error]&#xA;&#xA;</xsl:text>
    <xsl:text>&lt;xsl:value-of select="xs:integer(1) </xsl:text>
    <xsl:text>div xs:integer(0)"/&gt;&#xA;</xsl:text>
    <xsl:text>  [fatal error]&#xA;&#xA;</xsl:text>
    <xsl:text>&lt;xsl:value-of select="xs:decimal(1.0) </xsl:text>
    <xsl:text>div xs:decimal(0.0)"/&gt;&#xA;</xsl:text>
    <xsl:text>  [fatal error]&#xA;&#xA;</xsl:text>
    <xsl:text>&lt;xsl:value-of select="xs:double(1.0) </xsl:text>
    <xsl:text>div xs:double(0.0)"/&gt;&#xA;  </xsl:text>
    <xsl:value-of
```

```
      select="(xs:double(1.0) div xs:double(0.0), '&#xA;&#xA;')"/>
    <xsl:text>&lt;xsl:value-of select="xs:float(1.0) </xsl:text>
    <xsl:text>div xs:float(0.0)"/&gt;&#xA;  </xsl:text>
    <xsl:value-of
      select="(xs:float(1.0) div xs:float(0.0), '&#xA;&#xA;')"/>

    <xsl:text>Dividing durations by zero:&#xA;&#xA;</xsl:text>
    <xsl:text>&lt;xsl:value-of select="xs:yearMonthDuration</xsl:text>
    <xsl:text>('P3Y4M') div xs:double(0.0)"/&gt;&#xA; </xsl:text>
    <xsl:text>  [fatal error]&#xA;&#xA;</xsl:text>
    <xsl:text>&lt;xsl:value-of select="xs:yearMonthDuration</xsl:text>
    <xsl:text>('P3Y4M') div &#xA;                      </xsl:text>
    <xsl:text>xs:yearMonthDuration('P0Y0M')"/&gt;&#xA; </xsl:text>
    <xsl:text>  [fatal error]&#xA;&#xA;</xsl:text>
    <xsl:text>&lt;xsl:value-of select="xs:dayTimeDuration</xsl:text>
    <xsl:text>('P3DT8H') div xs:double(0.0)"/&gt;&#xA; </xsl:text>
    <xsl:text>  [fatal error]&#xA;&#xA;</xsl:text>
    <xsl:text>&lt;xsl:value-of select="xs:dayTimeDuration</xsl:text>
    <xsl:text>('P3DT8H') div &#xA;                      </xsl:text>
    <xsl:text>xs:dayTimeDuration('P0DT0H')"/&gt;&#xA; </xsl:text>
    <xsl:text>  [fatal error]&#xA;&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

As we noted before, dividing `xs:double` or `xs:float` by zero returns `INF`; everything else causes a fatal error. Dividing `xs:yearMonthDuration` or `xs:dayTimeDuration` by zero or by an interval of length zero also causes a fatal error.

It's possible division by zero will be a static error, but it's far more likely to occur at runtime. If you migrate your stylesheet to use XSLT 2.0's math processing, you'll need to test your code to make sure any division by zero error cases are handled correctly.

Type Checking Is Much Stricter

XSLT 1.0 had a very simple type system. The support for XML Schema in XSLT 2.0 means type checking is much stricter. For example, in XSLT 1.0, this statement works:

```
<xsl;value-of select="substring-before(12345, 3)"/>
```

This returns the value `12`, which causes a static error in XSLT 2.0. To get this function to work in XSLT 2.0, we have to write it like this:

```
<xsl;value-of select="substring-before(string(12345), string(3))"/>
```

As you migrate your stylesheets from XSLT 1.0 to 2.0, you'll find yourself explicitly casting values such as this quite often.

Calling Some Functions with More Than One Node Causes an Error

In XSLT 1.0, there were a number of functions that took a node as an argument. If we passed more than one node to the function, it simply took the first node and ignored

the rest. XSLT 2.0 is much more strict; passing more than one node to a function that requires a single node is a fatal error. You can certainly make the argument that using the argument `/purchase-order/items/item` when you really mean `/purchase-order/items/item[1]` is sloppy code. Whether you agree with that opinion or not, XSLT 2.0 forces you to pass a single node to the function.

The XSLT 1.0 and XPath 1.0 functions `generate-id( )`, `local-name( )`, `name( )`, `namespace-uri( )`, `number( )`, and `string( )` require a single node in XSLT 2.0. To avoid any problems, simply add the predicate `[1]` to your stylesheets as you migrate them. In addition, if you use an XPath expression to select nodes for the `concat( )` function, that XPath expression cannot return more than a single node. The expression `concat('#1. ', /report/brand/name)` throws an error in XSLT 2.0 if more than one node matches the XPath expression. On the other hand, `concat('#1. ', (/report/brand/name)[1])` always works.

This is a dynamic error. It can be difficult to catch if your XPath expressions typically select only one node. That means the error condition occurs in the rare case that the XPath expression returns more than one node. If you need only one node, changing your expression from `x/y/z` to `(x/y/z)[1]` is the best way to go.

Approaches to Migration

There are several approaches to migrating your stylesheets to XSLT 2.0. We'll outline them here.

Write (or Rewrite) Your Stylesheets from Scratch

Obviously this is the most drastic option. If you don't have a large investment in XSLT 1.0 code, you can start writing XSLT 2.0 stylesheets from scratch. It's more likely that you would replace a few stylesheets used in parts of your application, and then use your expertise and experience to rewrite larger and more sophisticated stylesheets.

This requires a significant commitment in development resources and training. Having the resources to develop stylesheets from scratch is one thing; those resources need to understand all of the new features of XSLT 2.0 and how to take advantage of them.

Change the Version to 2.0 and See What Happens

The simplest approach to migration is to change the `version` attribute of the `<xsl:stylesheet>` element to `2.0`. If you have any static errors, the XSLT 2.0 processor will flag those immediately. You can then fix the errors or use `version="1.0"` on specific XSLT elements so the XSLT 2.0 processor will use backwards-compatible mode.

If you use this approach, it's important to check for the dynamic errors we covered in the previous section. If you add `version="2.0"` to your stylesheet and it runs with an

XML document, that doesn't mean it will run with every document. As with any modified code, testing is crucial before you put your stylesheet into production.

Replace Awkward XSLT 1.0 Code with XSLT 2.0 Features

A less drastic way to migrate to XSLT 2.0 is to look through your stylesheets for templates that could be replaced with XSLT 2.0 features. For example, if you have a recursive template to do string replacement, replacing that with the new `replace( )` function will simplify your code. (It's likely it will perform better as well.) Anything that uses recursion or extension functions is a candidate for replacement with XSLT 2.0.

Another thing to look for is using variables extensively. If you use a variable to store the result generated by a named template, consider replacing the named template with an `<xsl:function>`. Using an XSLT 2.0 function means you can invoke the code inside an XPath expression. The function call can replace the variable and the named template with a single step.

If the templates in your stylesheet pass many parameters, consider using tunnel parameters. As stylesheets grow in complexity and power, it's common for every template to accept one or more parameters, and then pass those parameters on to any templates it invokes. The stylesheet needs to pass those parameters to every template in case they're needed somewhere. Tunnel parameters make your code much cleaner and simplify maintenance.

Mix XSLT 1.0 and XSLT 2.0 in the Same Stylesheet

A very useful technique for XSLT 2.0 migration is to process parts of the stylesheet in XSLT 1.0 or XSLT 2.0 mode. The simplest way to do this is to create an stylesheet with `version="2.0"`, adding `version="1.0"` to templates as needed. (Remember, the `version` attribute can be used on any XSLT element except `<xsl:output>`.) This lets you migrate your stylesheet in stages without worrying about dynamic errors. If your stylesheet depends on math being processed using XSLT 1.0 rules, you can set that section of the stylesheet to use `version="1.0"`. As your XSLT 2.0 experience grows, you can address the XSLT 1.0 sections of the stylesheet and migrate them to XSLT 2.0.

Don't Migrate at All

This is the cheapest and fastest solution. If your stylesheets aren't good candidates for using the new features of XSLT 2.0, there's no real reason to migrate. Having said that, it's unlikely that any sophisticated stylesheet can't be improved by any of the new features.

The architecture of your application can determine whether you should migrate. If you application sends XML data and an XSLT stylesheet down to a browser, that browser has to have an XSLT 2.0-compliant processor. There aren't any browsers that have XSLT 2.0 support built in, so it could be a while before migration is an option for you.

Glossary

A

absolute location path
: A location path that begins with /, followed by one or more location steps separated by /. All location paths that begin with / are evaluated from the root node, so they always return the same result, regardless of the context node. Compare with *location path* and *relative location path*.

ancestor
: A node that appears above a given node. Ancestors include a node's parent, its parent's parent, its parent's parent's parent, etc. XPath also defines the `ancestor` axis, which includes a node's parent, its parent's parent, its parent's parent's parent, etc., but not the node itself. Contrast with *parent*.

[2.0] arity
: The number of parameters required by a function.

[2.0] atomic value
: A value from one of the atomic datatypes defined by XML Schema (or a datatype derived from one of those types). `xs:integer`, `xs:dayTimeDuration`, and `xs:string` are examples of atomic datatypes.

attribute node
: The XPath node type that represents an attribute from an XML document. Attributes are different from other nodes because an attribute node is not considered a child of the element node that contains it. Despite this fact, the element node is considered the parent of the attribute node.

attribute set
: A named group of attributes. You can create an attribute set (with the `<xsl:attribute-set>` element), then reference that attribute set elsewhere. For more information, see the description of the `<xsl:attribute-set>` element in Appendix A.

attribute value template
: An expression that contains an XPath expression in curly braces ({}). The XPath expression is evaluated at runtime, and its value replaces the expression. For an example of an attribute value template, see the discussion of the `<xsl:attribute>` element in Appendix A; Chapter 3 contains a complete discussion of attribute value templates.

axis
: A relationship between the context node and other nodes in the document. XPath defines 13 different axes; see "Axes" in Chapter 3 for a complete discussion of them.

B

base URI
: The URI associated with every node in the XPath source tree. In certain circumstances, the base URI is used to resolve references to other resources. If a given node is an element or processing-instruction node and that node occurs in an external entity, then the base URI for that node is the base URI of the external entity. If an element or processing-

instruction node does not occur in an external entity, then its base URI is the base URI of the document in which it appears. The base URI of a document node is the base URI of the document itself, and the base URI of an attribute, comment, namespace, or text node is the base URI of that node's parent.

C

[2.0] cast
The conversion of a value from one atomic type to another.

CDATA section
A section of an XML document in which all markup is ignored. A CDATA section begins with the characters `<![CDATA[`, and ends with the characters `]]>`. If two right brackets appear in the content of a CDATA section, they must be escaped. Within a stylesheet, determining whether a given text node was originally a CDATA section is not possible. It is possible to generate certain elements as CDATA sections with the `cdata-section-elements` attribute of the `<xsl:output>` element.

child
An immediate descendant of a given node. Contrast with *descendant*. `child` is also the name of one of the XPath axes. The children of a node include all comment, element, processing-instruction, and text nodes. Attribute nodes and namespace nodes are not considered children.

[2.0] codepoint
A number that represents a Unicode character. For example, the number `x20AC` is the codepoint for the Euro sign (€).

[2.0] codepoint collation
The collation that simply compares strings according to their codepoint values. This is the default collation used by all XSLT processors.

[2.0] collation
A set of rules defining how string values should be compared. In XSLT 2.0 and XPath 2.0, processors have the option to support special collations that allow comparisons based on various languages. For example, the German word for street can be spelled *Strasse* or *Straße*; a custom collation might define those two strings as equal.

comment node
A node that represents a comment from the original XML document. This is one of the seven kinds of nodes defined by XPath.

[2.0] complex type
Any datatype defined with XML Schema that can have children or attributes.

[2.0] constructor function
A function used to create an atomic value of a particular datatype. The name of the function is the name of the datatype; the function `xs:integer( )` is a constructor function for the `xs:integer` datatype.

context
A data structure that determines how XPath expressions are evaluated.

[1.0] In XPath 1.0, the context consists of five items: the context node, a pair of nonzero positive integers (the context position and context size), a set of variable bindings, a function library, and the set of namespace declarations that are in scope.

[2.0] In XPath 2.0, the context includes everything from the XPath 1.0 context plus several other items. These include the datatypes and namespaces currently in scope. See "The XPath Context" in Appendix B for a complete description of everything in the context.

[2.0] context item
The item from which all XPath expressions are evaluated. In XPath 2.0, if the context item is a node, the context node and the context item are the same. If the context item is an atomic value, the context node is undefined.

context node
The node from which all XPath expressions are evaluated. The context node is analogous to the current directory at a command

prompt; all commands you type at a command prompt are evaluated in terms of the current directory. Compare with *current node*.

[2.0] In XPath 2.0, the context node is undefined if the context item is an atomic value. If the context item is a node, the context node and the context item are the same.

context position

[1.0] A nonzero positive integer that indicates the position of the current node. The context position is always less than or equal to the context size.

[2.0] A nonzero positive integer that indicates the position of the current item within the sequence of items currently being processed.

context size

[1.0] The number of nodes in the current node list.

[2.0] The number of items in the sequence of items currently being processed.

current node

[1.0] The node currently being processed. The node is defined by the `select` attribute of the `<xsl:apply-templates>` or `<xsl:for-each>` elements. Except within a predicate expression, the current node and the context node are the same.

[2.0] The item currently being processed.

current node list

The list of nodes selected by the `select` attribute of the `<xsl:apply-templates>` or `<xsl:for-each>` element currently being processed. By default, the current node list is in document order, but it may be reordered with one or more `<xsl:sort>` elements.

[2.0] In XPath 2.0, the current node list is referred to as "the sequence of items currently being processed" in lieu of the term "current node list."

D

descendant

A given node's children, its children's children, its children's children's children, etc. `descendant` is also the name of one of the axes defined by XPath. Contrast with *child*.

document element

The element in the XML source document that contains the entire XML document. The node that represents the document element is a child of the root node; the root node and the element node for the document element are not the same.

document node

In a tree created from a well-formed XML document, the root node will be a document node with exactly one element node as its child. That element node represents the document element, and does not have any text nodes as children. An XML parser will only create document nodes from well-formed XML documents. However, it is possible within an XSLT stylesheet to create a document node that is not well-formed. See the discussion of the `<xsl:document>` element for an example.

document order

The order in which a set of nodes appeared in the XML source document.

[2.0] dynamic context

The portion of the *context* that can change during the evaluation of a stylesheet or XPath expression. The context item, context size, and context position are all part of the dynamic context. See "[2.0] The XPath 2.0 context" in Chapter 3 for a complete description of everything in the context. Compare with [2.0] static context.

E

element node

An XPath node that represents an element from the XML source document.

empty sequence
A sequence that contains zero items.

encoding
A set of characters, referenced in the XML declaration to describe the characters used in a particular document. The range of values for encodings is defined in *http://www.ietf.org/rfc/rfc2278.txt*. The range of values supported by a given XML parser or XSLT processor varies.

expanded name
The complete name of an element or attribute, including the local name and a possibly null namespace URI.

extension element
An element in an XSLT stylesheet whose namespace prefix references an extension. The XSLT specification defines how extension elements are identified in the stylesheet, but does not specify how they are implemented. Extension elements are implemented with a piece of code that is referenced in the stylesheet; each XSLT processor defines how that code is invoked to handle the transformation of the extension element. See Chapter 8 for an extensive discussion of extension elements and extension functions.

extension function
A function whose namespace prefix references an extension. The XSLT specification defines how extension functions are identified in the stylesheet, but does not specify how they are implemented. Extension functions are implemented with a piece of code that is referenced in the stylesheet; each XSLT processor defines how that code is invoked to handle the invocation of the extension function. See Chapter 9 for an extensive discussion of extension elements and extension functions.

F

fallback processing
Processing designed to handle the absence of an extension element or an extension function gracefully. This processing is typically accomplished with the `element-available( )` or the `function-available( )` function. When either function returns false, a stylesheet can respond gracefully to the absence of the requested function. XSLT also defines the `<xsl:fallback>` element, which can be used when an extension element is not available.

final result tree
A tree that forms part of the final output of a transformation. Once created, the contents of the final result tree are not accessible within the stylesheet itself.

I

ID
One of the basic datatypes defined by the XML specification. (It is also defined as type `xs:ID` in XML Schema.) In an XML document, one attribute of an element can be declared to be of type `ID`; this means that the value of that attribute must be unique across all attributes of type `ID` for all elements in the document. No more than one attribute on a given element can be of type `ID`. Attributes of type `ID` are useful for generating cross-references with the `id( )` function. See Chapter 5 for an extensive discussion of the `ID`, `IDREF`, and `IDREFS` datatypes.

IDREF
One of the basic datatypes defined by the XML specification. (It is also defined as type `xs:IDREF` in XML Schema.) In an XML document, an attribute declared to be of type `IDREF` must have a value that matches an `ID` attribute elsewhere in the document. Attributes of type `IDREF` are useful for generating cross-references with the `id( )` function. See Chapter 5 for an extensive discussion of the `ID`, `IDREF`, and `IDREFS` datatypes.

IDREFS
One of the basic datatypes defined by the XML specification. (It is also defined as type `xs:IDREFS` in XML Schema.) In an XML document, an `IDREFS` attribute must contain one or more whitespace-separated values,

each of which matches an `ID` attribute elsewhere in the document. Attributes of type `IDREFS` are useful for generating cross-references with the `id( )` function. See Chapter 5 for an extensive discussion of the `ID`, `IDREF`, and `IDREFS` datatypes.

[2.0] item
: A value that is either an atomic value or a node.

K

key
: A key is similar to a database index. It has three components: a *name*, used to identify the key (specified with the `name` attribute of the `<xsl:key>` element); the *nodes*, which will be returned by the key (specified with the `use` attribute of the `<xsl:key>` element); and the *values*, used to search for things in the key (specified with the `match` attribute of the `<xsl:key>` element).

  The key `<key name="language-index" match="defn" use="@language"/>`, for example, defines a new key named `language-index`. Given a value for the `language` attribute, the key returns all `<defn>` elements whose `language` attributes match the given value. See "Branching Elements of XSLT" in Chapter 5 for a complete discussion of keys and how they are used.

L

literal result element (LRE)
: An element in an XSLT stylesheet that does not belong to the XSLT namespace and is not an extension element. Literal elements are simply copied to the result tree.

local name
: The nonqualified portion of an element or attribute name. For example, in the element `<xsl:template>`, `template` is the local name.

location path
: An XPath expression that selects a set of nodes relative to the context node. Compare with *absolute location path* and *relative location path*.

location step
: Consists of three parts: an axis name, a node test, and zero or more predicate expressions. There are three location steps in the following XPath expression: `preceding-sibling::region/product[@name="Sandpiper"]/text( )`.

  The first location step is `preceding-sibling::region`; it has an axis name of `preceding-sibling` and a node test of `region`. It selects all `<region>` elements that are preceding siblings of the context node. It does not have a predicate expression.

  The second location step, which is `product[@name="Sandpiper"]`, has an axis name of `child`, the default axis. Its node test is `product` and it has the predicate `[@name="Sandpiper"]`. It selects all `<product>` children of the previous location step that have an attribute named `name` with a value of `Sandpiper`.

  The third location step, `text( )`, has an axis name of `child` and a node test of `text( )`. It selects all text node children of the previous location step. It does not have a predicate expression.

M

mode
: An XSLT feature that allows an element to be processed multiple times, using a different template and producing a different result each time. See the discussion of the `<xsl:apply-templates>` element in Appendix A for a detailed example of modes.

N

namespace
: A collection of element and attribute names that are associated with a URI.

namespace declaration
: Part of an XML document that defines a namespace for a particular part of the document. Namespace declarations appear as XML attributes. A namespace declaration that begins `xmlns=` defines the default

namespace, while a declaration that begins `xmlns:prefix` defines a namespace with a prefix.

namespace node
The XPath node type that corresponds to a namespace declaration in an XML document.

namespace prefix
Part of a qualified name used to associate an element or attribute with a namespace URI.

namespace URI
The URI associated with a collection of element and attribute names.

NCName
An XML noncolonized name, used for local names and namespace prefixes. An NCName must start with a letter or an underscore (_).

node test
An XPath expression that selects certain nodes. The expressions `child::*`, `para`, and `@id` select all child nodes, all `<para>` child nodes, and any attribute named `id`, respectively. Four node tests—`text( )`, `comment( )`, `node( )`, and `processing-instruction( )`—look like functions, even though they technically aren't. *[2.0]* XSLT 2.0 also includes the `document-node( )` node test.) See "XPath Node Tests" in Appendix B for a complete listing of XPath node tests.

O

output escaping
The process of changing reserved characters (such as `<`, `>`, and `&`) into their entity references (such as `<`, `>`, and `&`).

P

parameter
An XSLT mechanism used to bind a name to a value, defined with the `<xsl:param>` element. The difference between a parameter and a variable is that the value specified in the definition of a parameter is a default value. When the template or stylesheet that contains the parameter is invoked, the default value can be overridden. Like variables, though, once the value of a parameter is set, it cannot be changed.

parent
A node that appears immediately above a given node. A parent is a node's first ancestor. XPath also defines the `parent` axis, which contains a node's parent. With the exception of the root node, all nodes in an input document have a parent. Contrast with *ancestor*.

path expression
An expression that selects nodes from a tree. The expression is composed of some number of *steps*. The nodes returned by a path expression appear in *document order* with any duplicates removed.

pattern
A condition that a given node may or may not match. The syntax for a pattern is a subset of the syntax for XPath expressions. Patterns are used in the XSLT elements `<xsl:template>`, `<xsl:key>`, `<xsl:number>`, and `<xsl:for-each-group>`.

predicate expression
An XPath expression that appears in square brackets ([ ]). Predicate expressions filter a *[1.0]* node-set or *[2.0]* sequence, selecting only nodes that match the expression in square brackets. See "Predicates" in Chapter 3 for more information.

prefix
An abbreviation for *namespace prefix*.

[2.0] primitive type
XML Schema defines 19 primitive types. In the XPath 2.0 data model, these are all defined as subtypes of `xs:anyAtomicType`. (The 19 primitive types are `xs:boolean`, `xs:string`, `xs:decimal`, `xs:double`, `xs:float`, `xs:QName`, `xs:anyURI`, `xs:hexBinary`, `xs:base64Binary`, `xs:date`, `xs:time`, `xs:dateTime`, `xs:gYear`, `xs:gYearMonth`, `xs:gMonth`, `xs:gMonthDay`, `xs:gDay`, `xs:duration`, and `xs:NOTATION`.) XPath also adds `xs:untypedAtomic`, the datatype of any value

that has not been validated against a schema or cast to a primitive type.

processing instruction
Part of an XML document containing instructions for applications. Here is a sample processing instruction:

```
<?xml-stylesheet href="docbook/html/
   docbook.xsl" type="text/xsl"?>
```

This processing instruction associates an XSLT stylesheet with an XML document. See "Document Object Model (DOM)" in Chapter 1 for a complete discussion of processing instructions.

processing-instruction node
The XPath node type that represents a processing instruction from an XML document.

Q

qualified name
An element or attribute name that may be qualified with a namespace prefix. The format of a qualified name is `prefix:localname`, where the optional `prefix` and `localname` are both NCNames. In an XSLT stylesheet, `<xsl:template>` is a prefixed qualified name, while `<template>` is not. The names in an XSLT stylesheet (variable names, template names, mode names, etc.) are qualified names, whether they have prefixes or not. Note that if the default namespace is null (`xmlns=""`), an unprefixed QName will not have a namespace.

QName
An abbreviation for *qualified name*.

R

[2.0] regular expression
A pattern used to analyze strings. Regular expressions can be used in the XPath 2.0 functions `matches( )`, `replace( )`, and `tokenize( )`, as well as the XSLT 2.0 instruction `<xsl:analyze-string>`.

relative location path
A location path that consists of one or more location steps separated by `/`, but not starting with `/`. Compare with *location path* and *absolute location path*.

[1.0] result-tree fragment
A fragment of the result tree that can be associated with a variable in XSLT 1.0. See "XPath 1.0 Datatypes" in Appendix B for a more complete discussion of result-tree fragments. *[2.0]* In XSLT 2.0, the result-tree fragment has been replaced with the sequence.

root node
The XPath node that represents the root of an XML document. Note that the root node is not the same as the element node for the document element. The root node is specified with the XPath expression `/`. The children of the root node are the element node for the document element, as well as any comments or processing instructions that occur outside the document element. *[2.0]* In XSLT 2.0, root nodes that represent an XML document are called document nodes.

S

sequence
An ordered collection of *items*.

sibling
Two nodes that have the same parent. XPath defines the `preceding-sibling` and `following-sibling` axes.

[2.0] static context
The portion of the *context* that is known before an XML source document is processed. See "[2.0] The XPath 2.0 context" in Chapter 3 for a complete description of everything in the context. Compare with [2.0] dynamic context.

static error
An error that can be detected before an XML source document is processed.

step
A portion of a path expression that navigates from one node to another. An axis step, a node test, and a predicate are all steps in an path expression.

stylesheet
: An XML document, written with the XSLT vocabulary, that specifies how an XML document should be transformed.

T

template
: A rule in an XSLT stylesheet that defines how part of an XML document should be transformed. Templates are defined with the `<xsl:template>` element.

text node
: A group of characters from an XML document. Text nodes are one of the seven types of nodes defined by XPath. The XPath specification states that as much text as possible must be combined into a single text node. In other words, a text node will never have a preceding or following sibling that is also a text node.

top-level element
: An element whose parent is the `<xsl:stylesheet>` element.

[2.0] tunnel parameter
: Within a template, a tunnel parameter is automatically passed on to any templates called by that template. The tunnel parameter is passed on recursively to any further templates that may be invoked along the way.

[2.0] type annotation
: The datatype associated with every element and attribute node. Type annotations are added to element and attribute nodes when they are validated by a schema. Any nodes that are not validated with a schema have the datatype `xs:untyped`; any attribute that is not validated has the datatype `xs:untypedAtomic`.

U

unparsed entity
: A resource in an XML document whose contents may or may not be text, and if text, may not be XML. Every unparsed entity has an associated XML notation. See the discussion of the `unparsed-entity-uri( )` function in Appendix C for more details on unparsed entities.

[2.0] unparsed text
: Text that is returned by the `unparsed-text( )` function. As the name implies, this text is not parsed in any way, so it may or may not be valid XML, well-formed XML, or any kind of markup at all.

URI
: Uniform Resource Identifier, a generalized version of the URLs used on the Web. URIs are defined by RFC 2396 and only support roughly 60 ASCII characters. Internationalized Resource Identifiers (IRIs), defined by RFC 3987, are supported by XSLT 2.0 and support a much wider range of characters. The XPath 2.0 `iri-to-uri( )` function converts an IRI to a URI as a convenience.

V

valid document
: An XML document that follows the basic rules of XML documents and additionally follows all rules of its associated document type definition or schema. See "XML Document Rules" in Chapter 1 for a complete discussion of the XML document rules; "Document Type Definitions (DTDs) and XML Schemas," also in Chapter 1, discusses document type definitions and schemas.

validation
: The process of comparing a node against a schema. If the node is valid, that node and all its children and attributes are assigned a datatype based on the node definitions in the schema.

variable
: An XSLT mechanism used to bind a name to a value, defined with the `<xsl:variable>` element. Variables are different from parameters because parameters can have default values. One significant difference between XSLT variables and variables in most other programming languages is that once an XSLT variable is bound, its value

cannot be changed. See "Variables" in Chapter 5 for a complete discussion of the `<xsl:variable>` element and how it is used.

W

well-formed document
: An XML document that follows the basic rules of XML syntax. See "XML Document Rules" in Chapter 1 for a complete discussion of those rules.

whitespace
: One of four characters: space (`&#x20`), tab (`&#x09`), return (`&#x0D`), or linefeed (`&#x0A`).

X

XML declaration
: Part of an XML document that defines the version of XML being used and the encoding of the document. Although the XML declaration looks like a processing instruction, it is not. For that reason, you cannot access the XML declaration from an XSLT stylesheet or an XPath expression.

Index

Symbols

!= (not equal) operator, 556
all mode attribute value, 159
current mode attribute value, 159
default mode attribute value, 159
$ (dollar sign) metacharacter, 906
((left parenthesis) metacharacter, 906
) (right parenthesis) metacharacter, 906
* (asterisk)
    as attribute, 52
    metacharacter, 906
    multiplication operator, 556
    occurrence indicator operator, 556
    quantifier character, 900
    wildcard, 51, 52, 61
+ (plus sign)
    addition operator, 556
    metacharacter, 906
    occurrence indicator operator, 556
    quantifier character, 900
    unary plus operator, 556
, (comma) sequence operator, 557
- (hyphen) metacharacter, 906
- (minus sign)
    subtraction operator, 557
    unary minus operator, 557
. (period) metacharacter, 905
/ (slash)
    location step operator, 557
    operator, 77
// (double slash)
    comment, 309
    location step operator, 557
    operator, 62
< (less than) operator, 146, 557
<< (node-before) operator, 102, 557
<= (less-than-or-equal-to) operator, 146, 557
= (equal to) operator, 557
> (greater than) operator, 558
>= (greater than or equal to), 558
>> (node-after) operator, 101, 558
? (question mark)
    metacharacter, 906
    occurrence indicator operator, 558
    quantifier character, 900
@* (at-sign and asterisk) wildcard, 61
[(square bracket) metacharacter, 906
[ ] (square brackets) operator, 558
\ (backslash) metacharacter, 906
\(left parenthesis escape character, 908
\) right parenthesis escape character, 908
* asterisk escape character, 908
\+ plus sign escape character, 908
\- hyphen or minus escape character, 908
\. period escape character, 908
\? question mark escape character, 908
\c escape character, 909
\C escape character, 909
\d escape character, 908
\D escape character, 908
\i escape character, 909
\I escape character, 909
\n newline escape character, 907
\p escape character, 909
\P escape character, 909
\r return escape character, 907
\s escape character, 908
\S escape character, 908
\t tab escape character, 907

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

\w escape character, 908
\W escape character, 909
\[left square bracket escape character, 908
\\ backslash escape character, 907
\] right square bracket escape characters, 908
\{ left curly brace escape character, 908
\| vertical bar escape character, 907
\} right curly brace escape character, 908
] (square bracket) metacharacter, 907
^ (carat) metacharacter, 906
{ (left curly brace) metacharacter, 907
{} (curly braces)
  in regular expressions, 363
| (vertical bar) union operator, 98, 99, 558, 561, 906
} (right curly brace) metacharacter, 907
() (parentheses) parenthesized expression operator, 556

A

absolute expressions
  XPath, 58
absolute location path
  defined, 933
absolute XPath expressions, 62
abstract datatypes
  about, 888–890
abstract elements
  about, 888–890
abs() function, 569–570
accessor functions
  list of, 563
addition (+) operator, 556
addition operator
  XPath, 71
adjust-date-to-timezone() function, 570–572
adjust-dateTime-to-timezone() function, 573–575
adjust-time-to-timezone() function, 575–577
Altova XSLT engine
  about, xii
  commands for Hello World example, 26, 40
  global parameters, 156
  installing, 23
ancestor axis, 64, 549
ancestor-or-self axis, 64, 107, 549
ancestor::table[] predicate, 66
ancestors
  defined, 933
anchors, 903–904
and operator, 558
anonymous datatype, 875
APIs (see SAX (Simple API for XML))
arity
  defined, 933
as attribute, 52, 435, 489, 525, 540, 544
asterisk (*) as attribute, 52
asterisk (*) metacharacter, 906
asterisk (*) multiplication operator, 556
asterisk (*) occurrence indicator operator, 556
asterisk (*) quantifier character, 900
asterisk (*) escape character, 908
atomic values
  comparing, 83
  defined, 933
  group-adjacent attribute, 233
  sequence, 215–219
  XPath 2.0, 46, 52
attr interface, 13
attribute axis, 64, 548
attribute nodes
  about, 49
  built-in template rules, 32
  defined, 546, 933
attribute sets
  defined, 933
  <xsl:attribute-set> element, 383
attribute value templates
  collation for sorting, 289
  defined, 933
attributes
  common attributes, 361
  declaring, 192
  elements with, 873
  empty elements with, 872
  list of by element, 362–544
  output methods, 31
  quotes, 7
  selecting, 59
  XPath, 45
  <xsl:sort element>, 211
attribute() node test, 51, 547
avg() function, 577–581
axes, xviii
  (see also specific axes)
  defined, 933
  list of in XPath, 63

location paths, 62
XPath, 548

B

back-references
about, 904–905
backslash (\) metacharacter, 906
backslash (\\) escape character, 907
base datatype
xs:anyType datatype, 70
base URIs
defined, 933
document() function, 252
base-uri() function, 581–584
basic XSLT 2.0 processor, 104
Bean Scripting Framework (see BSF)
bgcolor attribute, 148, 153
binary octets
datatypes, 70
bindings (see variable bindings)
block escapes
character group, 912–917
boolean datatype, 67, 552
boolean functions
list of, 563
boolean operators
XPath, 82–87
boolean values
<xsl:if> element, 146
boolean() function, 584–588
branching elements, 145–150
<xsl:choose> element, 148
<xsl:for-each> element, 149
<xsl:if> element, 146
browser support
XSLT 2.0, xiii
BSF (Bean Scripting Framework)
SVG pie chart example, 303
XSLT extensions, 326–330
built-in datatypes of XML Schema, 46
byte-order-mark attribute, 482, 503

C

C character groups, 912
calendars
formatting codes, 923
carat (^) metacharacter, 906
Cascading Style Sheets (see CSS)
case-insensitive mode, 363
case-order attribute, 213, 511
case-sensitivity
attributes, 8
cast
defined, 934
cast as operator, 94, 558
castable as operator, 95, 383, 558
casting
between datatypes, 883
Cc character group, 912
CDATA nodes
text in, 50
CDATA section
defined, 934
cdata-section-elements attribute, 481, 503
ceiling() function, 588–590
Cf character group, 912
character attribute, 487
character groups, 909–917
character maps
<xsl:character-map> element, 391
characters, xviii
(see also character groups; escape characters; metacharacters; whitespace)
\c escape character, 909
child axis, 63, 548
child(ren)
axes, 63–65
defined, 934
elements, 49
root nodes and, 48
text, 13, 60
closing characters (Pe) character group, 911
Cn character group, 912
Co character group, 912
codepoint collation
defined, 934
codepoints
defined, 934
codepoints-to-string() function, 591–592
collation attribute, 195, 427, 449, 511
collation functions
list of, 569
collations
custom, 287–293
defined, 934
collection types (see statically known default collection type)

collections
XPath context, 550
collection() function, 271, 592–594
combining XML documents, 245–275
collection() function, 271
document() function, 245–257
doc() and doc-available() functions, 269–271
grouping, 257–259
lookup tables, 254
unparsed-text() and unparsed-text-available() functions, 272
XSLT 2.0, 260–268
comma (,) sequence operator, 557
comment interface, 13
comment nodes
about, 50
built-in template rules, 32
defined, 546, 934
selecting, 60
comments
in XPath expressions, 102
comment() node test, 60, 547
common functions
defined, 330
compare() function, 291, 595–597
comparing
atomic values, 83
expressions, 82
sequences, 84
text, 291
comparison operators
atomic values, 84
complex types
defined, 934
concat() function, 170, 597–599
conditional expressions
XPath 2.0, 88
connector characters (Pc) character group, 911
constructor functions
defined, 934
list of, 564
XPath 2.0, 92
contact nodes
axes, 63–65
contains() function, 599–602
content
separation from presentation, 5
context, xviii
(see also dynamic context; functions; namespace declarations; static context; variables)
defined, 934
nodes, 28
XPath, 55, 550
context functions
list of, 564
context items
defined, 57, 934
self axis, 64
XPath context, 550
context nodes
defined, 934
XPath context, 550
context position
defined, 935
XPath context, 550
context size
defined, 57, 935
XPath context, 550
control characters (Cc) character group, 912
Coordinated Universal Time (UTC), 69
copy-namespaces attribute, 401, 406
count attribute, 122, 466
count() function, 602–605
count() predicate, 66
cross-references (see links)
cross-referencing functions, 564
CSS (Cascading Style Sheets)
about, 1
compared to XSLT, 112
CSS styles
HTML documents, 399
curly braces ({})
in regular expressions, 363
curly braces escape characters, 908
currency symbols (Sc) character group, 912
current node lists
defined, 935
current nodes
defined, 935
current-dateTime() function, 610–611
current-date() function, 609–610
current-grouping-key() function, 228, 229, 231, 614–615
current-group() function, 228, 229, 611–614
current-time() function, 616

current() function, 605–609

D

d attribute, 315
D formatting code, 132
D1o formatting code, 132
dashes (Pd) character group, 911
data binding
  defined, 16
data model
  XPath, 46–54
data-type attribute, 211, 511
databases
  extension elements, 333–339
  Xalan XSLT processor, 337–339
datatype operators
  XPath 2.0, 93–97
datatypes, xviii
  (see also built-in datatypes; list types of XML Schema; union types; specific datatype)
  converting to boolean values, 146
  defining, 875–890
  for division, 927
  parameters, 158
  XML ID, IDREF and IDREFS, 181–194
  in XML schemas, 10
  XPath, 67–71, 551–555
  XPath 2.0, 926
  XSLT 2.0, 12, 162, 926
data() function, 617–620
date functions
  defined, 330
  list of, 565
dates
  formatting, 130–132, 921–923
dates-and-times:month-name() function, 332
dateTime() function, 621–622
day-from-dateTime() function, 623–625
day-from-date() function, 622–623
days-from-duration() function, 625–626
debugging, xviii
  (see also errors)
  stylesheets, 35
decimal digits (Nd) character group, 910
decimal numbers
  formatting, 127–130
decimal-separator attribute, 408
declarations (see namespace declarations; XML declarations)
declaring, xviii
  (see also defining)
  attributes, 192
  elements, 871–875
deep-equal() function, 626–631
default collation sequences
  URIs, 362
default namespace
  XPath context, 550
default values
  attributes, 49
  parameters, 155
default-collation attribute, 362, 521
default-collation() function, 631–632
default-validation attribute, 520
defining, xviii
  (see also declaring)
  datatypes, 875–890
depoint-equal() function, 590–591
descendant axis, 64, 549
descendant-or-self axis, 64
descendants
  defined, 935
design goals (see features)
digit attribute, 409
digits (see numbers)
disable-output-escaping attribute, 526, 528
distinct-values() function, 218, 632–637
div (division) operator, 558
division
  datatypes, 927
division operator, xviii
  (see also integer division (idiv) operator)
  XPath, 77
doc-available() function, 269–271, 638–639
DocBook documents
  converting to HTML, 163
  using group-starting-with attribute, 238
  <xsl:for-each-group> element, 431
doctype-public attribute, 481, 503
doctype-system attribute, 481, 503
document elements
  defined, 6, 935
document interface, 13
document nodes, xviii
  (see also root nodes)
  built-in template rules, 32
  defined, 935
Document Object Model (see DOM)

document order
defined, 935
document root elements
versus root elements, 105
Document Type Definition) (see DTD)
document-node() node test, 52, 547
document-uri() function, 642–643
documents, xviii
(see also combining XML documents; DocBook documents; HTML (HyperText Markup Language); statically known documents; unstructured XML documents; XML documents)
XPath context, 550
document() function, 245–257, 639–642
doc() function, 269–271, 637–638
dollar sign ($) metacharacter, 906
DOM (Document Object Model)
about, 19
combining data, 246
programming interface for, 12
using extension elements, 345
DOM trees
example, 14
dot-all mode, 363
double slash (//) comment, 309
double slash (//) location step operator, 557
double slash (//) operator, 62
DTD (Document Type Definition)
rules for, 8–10
versus XML schemas, 10
duplicate IDs
generate-id() function, 204
duration functions
list of, 565
Dwo formatting code, 131
dwo formatting code, 132
dynamic context
about, 57
defined, 935
XPath 2.0, 56
dynamic errors (see errors)
dynamic functions
defined, 330
dynamic type
defined, 96
dynamic typing
adding nodes, 73
dynamically scoped variables
compared to tunnel parameters, 166

E

EcmaScript code
SVG pie chart example, 304
element interface, 13
element nodes
about, 48
built-in template rules, 32
defined, 545, 935
element-available() function, 282, 643–646
elements, xviii
(see also branching elements; document elements; extension elements; LRE (literal result element); namespace-qualified elements; non-XSLT elements; tags; top-level elements; wrapper elements; specific element)
children, 49
declaring, 871–875
empty, 871–873
key() functions defined for, 197
list of, 362–544
with mixed content, 874
in scope of namespace declarations, 50
selecting text from, 59
with text, 873–874
top level, 33
versus tags, 11
whitespace in, 33
XML document rules, 6
elements attribute, 498, 517
element() node test, 51, 547
else keyword, 89
empty elements
creating, 871–873
empty sequences
defined, 936
empty tags
end markers, 8
empty() function, 646–648
enclosing marks (Me) character group, 910
encode-for-uri() function, 648–650
encoding
defined, 936
encoding attribute, 481
end tags
parsing, 8
ends-with() function, 650–652

entities (see unparsed entities)
eq (equal to) operator, 559
equal to (=) operator, 557
equal to (eq) operator, 559
errors, xviii
  (see also debugging; static errors)
  calling functions with multiple nodes, 929
  extension element prefixes, 282
  imported templates, 369
  migration, 926–928
  passing multiple sequences to generate-id() function, 203
  priority attributes on name templates, 525
  regular expression matches, 363
  variables at same level with same name, 533
  version attribute, 361
  <xsl:call-template> element with undefined parameter, 926
  XslCompiledTransform object, 300
error() function, 652–655
escape characters, xviii
  (see also block escapes; output escaping)
  list of, 907–909
  node-before (<<) operator, 102
  output documents, 526
escape-html-uri() function, 655
escape-uri-attributes attribute, 482, 504
event-driven interfaces
  versus in-memory interfaces, 16
events
  in SAX, 15
every operator, 559
every operators, 89
exactly-one() function, 656–658
examples
  testing of, xii
except operator, 98, 559
exclamation mark equal sign (!=) not equal operator, 556
exclude-result-prefixes attribute, 361, 520
EXIF tags, 359
exists() function, 658–659
expanded names
  defined, 936
expressions, xviii
  (see also absolute expressions; conditional expressions; node tests; path expressions; predicate expressions; quantified expressions; regular expressions; relative expressions: XPath expressions)
  comments in, 102
  comparing, 82
  datatypes returned by XPath 1.0, 67
EXSLT library, 330
  SVG pie chart example, 303
extending XSLT, 277–360
  accessing databases, 333–339
  creating new functions, 279–281
  custom collations, 287–293
  EXSLT library, 330
  hidden word graphics, 293–303
  multiple output files, 281–286
  photo album example, 339–359
  SVG pie charts, 303–326
  XSLT extension mechanism, 277–279
Extensible Markup Language (see XML)
Extensible Stylesheet Language (XSL)
  standards, 18
Extensible Stylesheet Transformations (XSLT)
  history, 1
extension
  creating datatypes by, 882–883
extension elements, 278
  accessing databases, 333–339
  defined, 936
  photo album example, 339–359
extension functions
  about, 278
  defined, 936
  EXSLT library, 330
  .NET framework, 354–359
extension-element-prefixes attribute, 282, 362, 520
external parameters (see global parameters)
ext() node test, 50

F

fallback processing, 278
  extension functions, 936
false() Function, 659
features
  XPath 2.0, 45
  XSLT, 2
files
  multiple output files, 281–286
final quote marks (Pf) character group, 911
final result trees

defined, 936
flags attribute, 363
floating-point numbers
datatype, 68, 552
floor() function, 660–662
FNn formatting code, 132
following axis, 65, 549
following-sibling axis, 64, 549
for loop processor, 174–179
for operator, 89, 559
formal semantics
XQuery 1.0 XPath 2.0, 17
format attribute, 121, 466, 502
format-dateTime() function, 130, 665–668
format-date() function, 130, 264, 662–665
format-number() function, 127–130, 252, 668–672, 920
format-time() function, 130, 672–674
formatting
dates and times, 130–132
decimal numbers, 127–130
formatting characters (Cf) character group, 912
formatting codes, 919–923
dates and times, 921–923
numbers, 920–921
formatting objects
tags in Hello World example, 39
Formatting Objects specification (see XSL (Extensible Stylesheet Language))
forwards-compatible mode, 286, 364
fragments (see result-tree fragments)
from attribute, 466
function signatures
defined, 57
function-available() function, 294, 312, 332, 674–677
functional programming languages
relationship to XSLT, 2
functions, xviii
(see also extension functions; specific function)
about, 2
calling with more than one node, 928
context, 56
creating, 279–281
listed alphabetically, 569–870
listed by category, 563–569
in predicates, 65
XPath context, 550
XQuery 1.0 and XPath 2.0, 17

G

gDay datatype, 70
ge (greater than or equal to) operator, 559
generate-id() function, 198–204, 677–680
getStateName() function, 260, 262
global parameters, 155–158
glossary, 933–941
gMonth datatype, 70
goals (see features)
graphics (see hidden word graphics; pie charts; SVG (Scalable Vector Graphics))
greater than (>) operator, 558
greater than (gt) operator, 559
greater than or equal to (>=) operator, 558
greater than or equal to (ge) operator, 559
Greenwich Mean Time, 69
<greeting> element, 31
group-adjacent attribute, 228, 426
group-adjacent grouping style, 232–238
group-by attribute, 228, 426
group-by grouping style, 229
group-ending-with attribute, 228, 427
group-ending-with grouping style, 241
group-starting-with attribute, 228, 426
group-starting-with grouping style, 238
grouping
by distinct values, 261
multiple documents, 257–259
grouping functions
list of, 564
grouping-separator attribute, 408, 468
grouping-size attribute, 468
groups, xviii
(see also character groups; substitution groups)
datatypes, 876
nodes, 219–227
XSLT 2.0, 228–242, 925
gt (greater than) operator, 238, 559

H

h01 formatting code, 132
Hello World example, 25–44
processing stylesheets, 27–30
stylesheet structure, 30–35

transforming into different file types, 36–43
transforming XML documents, 25–27
hidden word graphics
generating, 293–303
history
XML, 4
XSLT, 1–3
hours-from-dateTime() function, 680–681
hours-from-duration() function, 681–683
hours-from-time() function, 683
href attribute, 441, 447, 502
HTML (HyperText Markup Language)
about, 4
<code> element, 461
converting DocBook documents to, 163
CSS styles, 399
generating from XML, 115
generating with links, 188–192
Hello World example, 30
mastheads, 151
using <xsl:key> element, 226
using group-adjacent attribute, 232
using group-starting-with attribute, 238
whitespace in, 115
wrapper elements, 34
writing script code, 309
HTML tables
photo album example, 339–359
Hwo formatting code, 132
HyperText Markup Language (see HTML)
hyphen (-) metacharacter, 906
hyphen (\-) escape character, 908

I

i flag, 363, 902
id attribute, 520
ID datatype
defined, 936
limitations of, 192
identity transform stylesheet, 133
idiv (integer division) operator, 560
IDREF datatype, 181–194, 936
IDREFS datatype, 181–194, 936
idref() function, 188, 689–693
id() function, 184–188, 192, 684–689
if operator, 89, 237, 260, 560
if statement, 263
implicit-timezone() function, 693
importing
stylesheets, 33
XML schemas, 890–892
In XSLT 2.0
<xsl:sort> element, 512
in-memory interfaces
versus event-driven interfaces, 16
in-scope-prefixes() function, 694–696
include-content-type attribute, 482, 504
indent attribute, 481, 504
index-of() function, 263, 696–699
infinity attribute, 409
inherit-namespaces attribute, 402, 417
initial quote marks (Pi) character group, 911
input-type-annotations attribute, 521
inputs
by function, 569–870
insert-before() function, 699–703
installing
XSLT processors, 20–24
instance of operator, 93, 560
integer datatype, 68
integer division (idiv) operator, 79, 560
interactivity
SAX, 15
interfaces
for programming, 12–16
intersect operator, 98, 99, 560
invoking
templates by name, 151
iri-to-uri() function, 703–704
is operator, 100, 560
<item> elements, 262
items, xviii
(see also context items)
defined, 937
item() datatype, 52
iteration
defined, 3
iterators
over sequences, 89

J

Jacl interpreter
XSLT extensions, 329
Java programs, xviii
(see also Saxon Java extensions; Xalan Java extensions)
commands for Hello World example, 41

global parameters, 156
Hello World example, 39
java programs
hidden word graphics, 295–298
JavaScript engine
XSLT extensions, 329
JDBC data source
Xalan XSLT processor, 337
JRuby language
XSLT extensions, 328
Jython
XSLT extensions, 327

K

Kay, Michael, xvii
keys, 194–198
defined, 937
<xls:key> element, 33
keywords
XPath, 556–561
key() function, 194–198–204, 224, 229, 258, 705–707

L

L letters character group, 909
lang attribute, 213, 328, 329, 467, 511
lang() function, 707–709
last() function, 709
le (less than or equal to) operator, 560
left curly brace (\{) escape character, 908
left curly brace ({) metacharacter, 907
left parenthesis escape character, 908
left square bracket (\[) escape character, 908
less than (<) operator, 557
less than (lt) operator, 560
less than or equal to (<=)operator, 557
less than or equal to (le) operator, 560
less-than (<) operator, 146
less-than-or-equal (<=) operator, 146
letter-value attribute, 467
letters
numbers as, 910
letters character groups, 909
level attribute, 466
libraries (see EXSLT library; .NET library; Xalan SQL library)
<line> element, 59
line separator (Zl) character group, 911
line[] predicate, 66
link points
creating, 190
links, 181–204
key facility, 194–198
unstructured documents, 198–204
XML ID, IDREF and IDREFS datatypes, 181–194
list types
creating, 884
literal result element (LRE)
defined, 937
Ll character group, 910
Lm character group, 910
Lo character group, 910
local names
defined, 937
matching nodes with, 52
local-name-from-QName() function, 712–714
local-name() function, 710–712
location paths, 55–66
axes, 62
context, 55
defined, 937
predicates, 65
relative and absolute expressions, 58
selecting things other than elements, 59
simple, 58
wildcards, 61
location step (/) operator, 557
location step (//) operator, 557
location steps
defined, 937
lookup tables
combining XML documents, 254
<xsl:function> element, 262
loops
emulating, 174–179
use of, 3
lower-case() function, 714–715
lowercase letters (Ll) character group, 910
LRE (literal result element)
defined, 937
lt (less than) operator, 560
Lt character group, 910
Lu character group, 909

M

M character group, 910

m flag, 363, 902
m01 formatting code, 132
m1o formatting code, 132
main method
  root templates as, 29
maps (see character maps)
marks
  character groups, 910
master documents
  document() function, 246
match attribute, 59, 195, 449, 524
matches() function, 715–719
matching
  namespace nodes, 60
mathematical functions
  defined, 331
mathematical operations
  errors, 927
  without recursion, 262
mathematical operators
  XPath, 71–81
max() function, 719–723
Mc character group, 910
Me character group, 910
media-type attribute, 482, 504
memory
  SAX versus DOM, 15
messages (see warning messages)
metacharacters
  about, 905–907
method attribute, 480, 503
methods, xviii
  (see also main method; static methods; output methods)
  root templates as main method, 29
Microsoft MSXSL (see .NET framework)
migration, 925–931
  approaches to, 929–931
  errors, 926–928
  XSLT 2.0 and XPath 2.0, 925
minus sign (-) metacharacter, 906
minus sign (-) subtraction operator, 557
minus sign (-) unary minus operator, 557
minus sign (\-) escape character, 908
minus-sign attribute, 409
minutes-from-dateTime() function, 728–729
minutes-from-duration() function, 729–731
minutes-from-time() function, 731–732
min() function, 724–728
Mn character group, 910
MNn formatting code, 132
MNn,3-3 formatting code, 132
mod (modulus) operator, 561
mod operator, 80, 123
mode attribute, 152, 158, 372, 524
modes, xviii
  (see also forward-compatible mode; processing modes)
  built-in template rules, 32
  defined, 937
modifiers (see presentation modifiers)
modifiers (Lm) character group, 910
modifiers (Sk) character group, 912
modulus (mod) operator, 561
month-from-dateTime() function, 734–735
month-from-date() function, 732–733
month-name() function, 332
months-from-duration() function, 735–736
MSXSL XSLT processor (see .NET framework)
Muench method, 224, 229
multi-line mode, 363
multiplication (*) operator, 556
multiplication operator
  XPath, 76

N

N numbers character groups, 910
name attribute, 194, 377, 383, 387, 391, 408, 416, 435, 448, 455, 482, 489, 500, 524, 531, 540
names, xviii
  (see also expanded names; local names; qualified names; XML names)
  invoking templates by, 151
namespace attribute, 377, 416, 443
namespace axis, 65, 549
namespace declarations
  defined, 937
  elements in, 50
  XPath context, 550
namespace nodes
  about, 50, 60
  built-in template rules, 32
  defined, 546, 938
  matching, 51
  selecting, 60
namespace prefix
  defined, 938

wildcards in elements attribute, 498
namespace prefixes
namespace nodes, 61
wildcards, 518
namespace URIs
defined, 938
namespace URLs
matching namespaces, 61
namespace-qualified elements
wildcards, 142
namespace-uri-for-prefix() function, 742–744
namespace-uri-from-QName() function, 745–749
namespace-uri() function, 740–742
namespaces
about, 19
default, 550
defined, 937
for extensions, 279
matching based on namespace URL, 61
rules for, 11
XML schemas, 893–896
<xsl:function> element, 279
name() function, 737–740
namspace prefixes
wildcards for, 62
NaN attribute, 409
NCName
defined, 938
Nd character group, 910
ne (not equal to) operator, 561
nesting
elements, 7
.NET framework
about, xii
commands for Hello World example, 26, 40
global parameters, 156, 157
hidden work graphics, 298–303
installing, 22
photo album example, 354–359
.NET library
SVG pie chart example, 303
newline (\n) escape character, 907
nilled() function, 749–751
Nl character group, 910
No character group, 910
node functions
list of, 566
node interface, 13
node operators
XPath 2.0, 100–102
node tests
defined, 938
XPath, 50
node-after (>>) operator, 101, 558
node-before (<<) operator, 102, 557
node-name() function, 751–755
node-set
specifying first elements using predicate expressions, 201
node-set datatype
conversion to boolean values, 146
defined, 67, 551
node-set functions
list of, 567
nodes, xviii
(see also attribute nodes; CDATA nodes; context nodes; comment nodes; current nodes; document nodes; element nodes; namespace nodes; processing-instruction nodes; range expressions; root nodes; siblings; text nodes; whitespace-only nodes)
adding, 73
arguments in id() and idref() functions, 185
combining XML documents, 252
comparing, 98
context, 28
functions with multiple, 928
invoking generate-id() functions against, 203
processing absolute expressions, 59
sequence, 215–219
setting base URIs, 253
tests, 547
types of in XPath, 46–50, 545
<xsl:apply-templates> element, 372
node() node test, 50, 62, 547
non-extractive XML processing
defined, 16
non-XSLT elements, 114
None. attribute, 449
nonspacing marks (Mn) character group, 910
normalization-form attribute, 482, 504
normalize-space() function, 142–143, 755–757
normalize-unicode() function, 757–759

not equal (!=) operator, 556
not equal to (ne) operator, 561
not() function, 83, 759–761
number datatype
  conversion to boolean values, 146
  defined, 68, 552
  errors in math operations, 927
<number> elements, 96
numbering
  document parts, 118–127
numbers, xviii
  (see also decimal numbers)
  errors in math operations, 927
  formatting codes, 920–921
  in predicates, 65
  \d any digit escape character, 908
numbers (N) character groups, 910
numbers as letters (Nl) character group, 910
number() function, 761–764
numeric functions
  list of, 566
numeric values
  single quotes, 168

O

objects (see formatting objects)
occurrence indicator (*) operator, 556
occurrence indicator (+) operator, 556
occurrence indicator (?) operator, 558
omit-xml-declaration attribute, 481, 504
one-or-more() function, 764–766
opening (Ps) character group, 911
operations (see mathematical operations)
operators, xviii
  (see also boolean operators, datatype operators; mathematical operators; set operators)
  precedence in XPath, 562
  XPath, 71–102, 556–561
  XQuery 1.0 and XPath 2.0, 17
or operator, 561
order attribute, 213, 511
order of operations
  boolean operators in XPath, 83
ordinal attribute, 127, 468
other letters (Lo) character group, 910
other numeric characters (No) character group, 910
other punctuation marks (Po) character group, 911
output, 113–144
  dates and times, 130–132
  decimal numbers, 127–130
  by function, 569–870
  multiple files, 281–286
  numbering document parts, 118–125
  text, 113–118
  whitespace, 139–144
  <xsl:copy> and <xsl:copy-of> elements, 132–139
output escaping
  defined, 938
output methods
  <xsl:output> element, 30, 480–487
output view
  of XML documents, 105
output-version attribute, 504
override attribute, 435

P

P character groups, 911
P formatting code, 132
paragraph separator (Zp) character group, 911
parameters, xviii, 152–167
  (see also global parameters; tunnel parameters)
  defined, 938
  defining in templates, 152
  passing to templates, 154, 158
  <xsl:param> element, 33
  XSLT 2.0, 158–167
parent
  defined, 938
parent axis, 63, 548
parentheses escape characters, 908
parenthesis metacharacter, 906
parenthesized expression operator, 556
parents
  axes, 63–65
  element nodes and attribute nodes, 49
parsing
  multiple XML documents, 245–253
  stylesheets, 27
  whitespace-only nodes, 140
  XML documents, 6, 13
  and XPath, 45
partial XML schema validation, 11

path (see location paths; relative location path)
path expressions
  defined, 938
pattern matching
  about, 2
pattern-separator attribute, 409
patterns
  defined, 938
Pc character group, 911
Pd character group, 911
PDF files
  Hello World example, 38
Pe character group, 911
per-mille attribute, 409
percent attribute, 409
period (.) metacharacter, 905
period (\.) escape character, 908
Pf character group, 911
photo album example, 339–359
PI (processing instruction interface), 13
Pi character group, 911
pie charts
  SVG, 303–326
plus sign (+) addition operator, 556
plus sign (+) metacharacter, 906
plus sign (+) occurrence indicator operator, 556
plus sign (+) quantifier character, 900
plus sign (+) unary plus operator, 556
plus sign (\+) escape character, 908
Po character group, 911
polymorphism
  invoking templates, 151
position() function, 242, 766–768
preceding axis, 64, 549
preceding siblings
  grouping with <xsl:variable> element, 224
preceding-sibling axis, 64, 549
predicate expressions
  defined, 938
  specifying first elements from node-set or sequences, 201
predicates
  comparing values, 187
  XPath, 65
  <xsl:for-each> element, 215
prefix-from-QName() function, 769–771
prefixes, xviii
  (see also namespace prefixes)
  extension elements, 282
  xsl namespace, 11
presentation
  separation from content, 5
presentation modifiers
  about, 922
primitive datatypes
  XML Schema, 552
primitive types
  defined, 938
priority
  template rules, 31
priority attribute, 524, 525
processing, xviii
  (see also non-extractive XML processing)
  predicates, 65
  sequences, 91
  stylesheets versions 1.0 and 2.0, 286
processing instruction interface, 13
processing instructions
  defined, 939
processing modes
  about, 902–903
processing namespace prefixes, 12
processing-instruction nodes
  about, 50
  built-in template rules, 32
  defined, 546, 939
  selecting, 60
processing-instruction() node test, 51, 60, 547
processors (see Altova XSLT engine; .NET framework; Saxon XSLT processor; Xalan XSLT processor; XSLT 1.0 processors; XSLT 2.0 processors; XSLT processors)
programming
  interfaces for, 12–16
Ps character group, 911
pull interfaces
  versus push interfaces, 16
punctuation
  character groups, 911
push interfaces
  versus pull interfaces, 16

Q

QName functions
  list of, 566
QName() function, 771–773

qualified names, xviii
(see also xs:QName datatype)
defined, 939
quantified expressions
XPath 2.0, 89
quantifiers, xviii, 900
(see also reluctant quantifiers)
query languages (see XQuery 1.0)
query statements
Xalan XSLT processor, 338
question mark (?) metacharacter, 906
question mark (?) occurrence indicator operator, 558
question mark (?) quantifier character, 900
question mark (\?) escape character, 908
quote marks
character groups, 911
quotes
attributes, 7

R

random functions
defined, 331
range (to) operator, 561
range expressions
XPath 2.0, 91
recursion
avoiding using XSLT 2.0 processors, 250
combining XML documents, 250–252
defined, 3
mathematical operations without, 262
template rules, 32
using, 169–174
XSLT 2.0 and XPath 2.0, 925
recursive templates, 250
<redirect:write> element, 281
references (see back-references)
regex attribute, 363
regex-group() function, 774–776, 898
regular expression functions
defined, 331
list of, 567
regular expressions, 897–917
anchors, 903–904
back-references, 904–905
character groups, 909–917
defined, 939
escapes, 907–909
metacharacters, 905–907
processing modes, 902–903
quantifiers, 900
reluctant quantifiers, 900–902
simple expressions, 897
subexpressions, 898–900
XSLT 2.0 and XPath 2.0, 926
relative expressions
XPath, 58
relative location path
defined, 939
relative references
base URIs and document() function, 252
RelaxNG schema language
about, 19
reluctant quantifiers
about, 900–902
remove() function, 776–781
replace() function, 173, 781–786, 899, 901
required attribute, 490
required parameters
XSLT 2.0, 161
resolve-QName() function, 786–788
resolve-uri() function, 788–790
restriction
creating datatypes by, 880–882
result tree fragment datatype, 552
result-prefix attribute, 458
result-tree fragments
defined, 939
return (\r) escape character, 907
reverse() function, 790–792
RFC 2396 and RFC2732
xs:anyURI datatype, 70
right curly brace (\}) escape character, 908
right curly brace (}) metacharacter, 907
right parenthesis escape character, 908
right square bracket (\]) escape characters, 908
root elements, xviii
(see also document elements)
versus document root elements, 105
root nodes, xviii
(see also document nodes)
about, 48
ancestor axis, 64
defined, 545
result-tree fragment, 935
root templates
as main method, 29

root() function, 792–795
rotate
  operator, 315
round-half-to-even() function, 798–799
round() function, 795–797
rules
  DTDs, 8–10
  templates, 31
  XML documents, 6–12

S

s flag, 363, 902
S symbols character groups, 911
s:float datatype, 69
SAX (Simple API for XML)
  about, 19
  programming interface for, 15
Saxon Java extensions
  photo album example, 347–353
Saxon XSLT processor
  about, xii
  collation for sorting, 289
  commands for Hello World example, 26, 40
  custom collations, 287
  databases, 334
  global parameters, 156
  installing, 21
  passing global parameters, 492
  undefined parameters, 160
Sc character group, 912
Scalable Vector Graphics (see SVG)
schema-aware XSLT 2.0 processors, 104
schema-element() node test, 51, 547
schema-location attribute, 443
schemas (see XML schemas)
Schematron
  about, 19
Schema] type attribute, 378
Schema] validation attribute, 378
scope
  elements in namespace declarations, 50
  parameters, 155
  variables, 169
seconds-from-dateTime() function, 799–800
seconds-from-duration() function, 801–802
seconds-from-time() function, 802–803
select attribute, 105, 127, 211, 216, 362, 372, 378, 381, 398, 405, 424, 426, 453, 455, 468, 489, 494, 500, 508, 511, 528, 531, 540
selecting
  attributes, 59
  comment nodes, processing instruction nodes and namespace nodes, 60
  text from elements, 59
self axis, 63, 548
semantics (see formal semantics)
separation
  of content and presentation, 5
separator attribute, 117, 378, 381, 529
separators
  character groups, 911
sequence (,) operator, 557
sequence functions
  list of, 567
sequence of items currently being processed, xviii
  (see also current node lists)
  defined, 935
sequences, xviii
  (see also default collation sequences; empty sequences)
  comparing, 84
  defined, 939
  groups of nodes or atomic values, 215–219
  iterators over, 89
  passing multiple to generate-id() functions, 203
  specifying first elements using predicate expressions, 201
  testing, 91
  XPath 2.0, 45, 46, 52
  <xsl:perform-sort> element, 216, 496
serialization
  XSLT 2.0 and XQuery 1.0, 17
set functions
  defined, 331
set operators
  XPath 2.0, 97–100
sets (see attribute sets)
SGML (Standard Generalized Markup Language), 4
siblings, xviii
  (see also preceding siblings)
  defined, 939

signatures (see function signatures)
Simple API for XML (see SAX)
simple expressions, 897
single quotes
  around values, 153
  numeric values, 168, 534
Sk character group, 912
slash (/) location step operator, 557
slash (/) operator
  XPath, 77
Sm character group, 911
some operator, 89, 561
sonnet[] predicate, 66
sorting
  data, 205–215
  document() function, 254
  using custom collations, 288–290
"sounds great, maybe later" (see SGML (Standard Generalized Markup Language))
space characters (Zs) character group, 911
spacing combining marks (Mc) character group, 910
SQL statements
  transforming from XML documents, 169
SQL support
  Saxon XSLT processor, 334
<sql:close> element, 334, 335
sql:close() function, 337–339
<sql:connect> element, 335
<sql:connection> element, 334
sql:disableStreamingMode() function, 337–339
sql:new() function, 339
<sql:query> element, 334, 335
sql:query() function, 337–339
square bracket escape characters, 908
square bracket metacharacters, 906
square brackets ([]) operator, 558
square brackets escape characters, 908
stable attribute, 512
standalone attribute, 481, 504
Standard Generalized Markup Language (SGML), 4
standards
  for stylesheets, 1
  XML, 18–20
  XSLT, 16
start (Ps) character group, 911
starts-with() function, 803–806
static context
  about, 56
  defined, 935
  XPath 2.0, 56
static datatype
  defined, 96
static errors
  defined, 939
static methods
  SVG pie chart example, 303
static-base-uri() function, 806–808
statically known default collection type
  defined, 57
statically known documents
  defined, 57
steps, xviii
  (see also location steps)
  defined, 939
<store> element, 310
string attribute, 487
string datatype, 68, 552
  conversion to boolean values, 146
string functions
  defined, 331
  list of, 568
string replacement
  using recursion, 169
string values
  in processing instructions, 50
string-join() function, 813–814
string-length() function, 815–817
string-to-codepoints() function, 817
strings
  replacing using recursion, 171
  in XPath boolean expressions, 82
string() function, 808–812
<strip-space> element, 142
stylesheet-prefix attribute, 458
stylesheets
  as XML documents, 2
  associating with XML documents, 19
  debugging, 35
  defined, 940
  importing, 33
  migration, 929
  processing in Hello World example, 27–30
  processing versions 1.0 and 2.0, 286
  standards, 1, 18

structure of in Hello World example, 30–35
transforming Hello World example, 25–27
XPath view of XML documents, 105–112
subexpressions, 898–900
subsequence() function, 273, 818–821
substitution groups
defined, 886
substring-after() function, 170, 824–826
substring-before() function, 170, 826–828
substring() function, 821–824
subtraction operator
XPath, 75
subtraction operator (-) operator, 557
sum() function, 250–252, 829–833
SVG (Scalable Vector Graphics)
about, 20
pie charts, 303–326
SVG files
Hello World example, 36
<svg:g> group element, 316
symbols
character groups, 911
syntax
escaping single quotes in XSLT 2.0 and XPath 2.0, 174
by function, 569–870
grouping, 228–242
tunnel parameters, 167
XPath, 45, 63
XQuery 1.0, 17
xsl:sort element, 209
system-property() function, 833–838

T

tab (\t) escape character, 907
tables (see HTML tables; lookup tables)
tags, xviii
(see also elements; EXIF tags)
formatting objects in Hello World example, 39
versus elements, 11
XML rules for, 8
<td> element, 380
templates, xviii
(see also attribute value templates; recursive templates; root templates)
defined, 940
defining parameters in, 152
emulating loops, 174–179
invoking by name, 151
passing parameters to, 154
rules, 31
XPath expressions, 59
terminate attribute, 453
test attribute, 147, 439, 536
testing, xviii
(see also node tests)
sequences, 91
text, xviii
(see also unparsed text)
adding white space to output, 143–144
in CDATA nodes, 50
children, 60
comparing, 291
elements with, 873–874
generating, 113–118
selecting from elements, 59
text interface, 13
text nodes
about, 49
built-in template rules, 32
defined, 546, 940
text() node test, 60, 547
time functions
defined, 330
list of, 565
times
formatting, 130–132, 921–923
timezone-from-dateTime() function, 839–840
timezone-from-date() function, 838–839
timezone-from-time() function, 840–841
titlecase (Lt) character group, 910
to (range) operator, 149, 561
tokenize() function, 273, 841–845
top-level elements
about, 33
defined, 940
toRadians() function, 332
trace() function, 845–848
transform attribute, 315
translate operator, 315
translate() function, 848–850
treat as operator, 96, 561
trees (see DOM trees; final result trees; result tree fragment datatype)
true() function, 851
tunnel attribute, 490, 540

tunnel parameters
defined, 159, 940
XSLT 2.0, 163–167
type annotation datatype, 940
type attribute, 402, 406, 412, 417, 502
type checking
XSLT 2.0, 928
type-available() function, 852–853
types, xviii
(see also primitive types; statically known default collection type)
nodes in XPath, 46–50

U

unabbreviated syntax
XPath, 63
unary minus (-) operator, 557
unary minus (–) operator, 81
unary plus (+) operator, 81, 556
undeclare-prefixes attribute, 482, 504
undefined parameters
XSLT 1.0, 159
Unicode
case-insensitive mode, 363
Uniform Resource Identifier (see URI)
union (|) operator, 98, 99, 558, 561, 906
union types
creating, 885
unordered() function, 853–855
unparsed entities
defined, 940
unparsed text
defined, 940
unparsed-entity-public-id() function, 855–856
unparsed-entity-uri() function, 857–858
unparsed-text-available() function, 272, 860–862
unparsed-text() function, 272, 858–860
unstructured XML documents
links, 198–204
upper case (Lu) character group, 909
upper-case() function, 862–864
ur-type datatype (see base datatype)
URI (Uniform Resource Identifier), xviii
(see also base URIs)
default collation sequences, 362
defined, 940
use attribute, 449
use-attribute-sets attribute, 383, 401, 416
use-character-maps attribute, 391, 482, 504
use-when attribute, 362
UTC (Coordinated Universal Time), 69

V

valid documents
defined, 940
validation
defined, 940
well-formed versus valid documents, 10
XML schemas, 11
validation attribute, 402, 406, 413, 417, 502
value attribute, 466
values, xviii
(see also atomic values; default values; string values)
of attributes, 7
comparing using predicates, 187
dates, 70
floating point, 69
grouping by, 261
IDs, 194
matching elements using key() function, 197
single quotes around, 153
variable bindings
XPath context, 550
variables
about, 2, 167
containing nodes, 536
context, 56
defined, 940
error with, 533
match and use attributes, 195
named templates, 930
version attribute, 361, 481, 520
vertical bar (\|) escape character, 907
vertical bar (|) union operator, 98, 99, 558, 561, 906
views (see output view)
VRML files
Hello World example, 41

W

warning messages
<xsl:style> element, 30
well-formed documents
defined, 941

versus valid documents, 10
whitespace
defined, 941
in DOM trees, 15
elements and, 33
in HTML, 115
processing, 139–144
<xsl:analyze-string> element, 364
<xsl:preserve-space> element, 498
whitespace (\s) escape character, 908
whitespace-only nodes
parsing, 140
width attribute, 153
wildcards
elements attribute, 498
location paths, 61
namespace prefixes, 518
namespace-qualified elements, 142
word (\w) escape character, 908
word graphics, hidden, 293–303
wrapper elements
HTML documents, 34

X

x flag, 364, 903
Xalan Java extensions
photo album example, 341–359
Xalan SQL library
accessing databases, 338
Xalan XSLT processor
<redirect:write> element, 281
about, xii
commands for Hello World example, 26, 40
database, 337–339
global parameters, 156
installing, 20
passing global parameters, 492
undefined parameters, 160
XDM (XPath 2.0 Data Model), 17
Xlink (XML linking language)
about, 20
XML (Extensible Markup Language), 4–12
history, 4
programming interfaces for, 12–16
rules, 6–12
standards, 18–20
XSLT standards, 16
XML 1.0
<aml:lang> and <xml:space> attributes, 49
rules, 6
XML declarations
defined, 941
rules for, 8
XML documents, xviii
(see also CDATA section; combining XML documents; document elements; namespace declarations; processing instructions; stylesheets; unparsed entities; unstructured XML documents; valid documents; well-formed documents; valid documents; well-formed documents)
root elements versus XPath document root elements, 105
stylesheets as, 2
transforming into SQL statements, 169
XPath view of, 104–112
XML ID datatype, 181–194
XML linking language (Xlink), 20
XML names
\i escape character, 909
xml namespace prefix, 108
XML Path Language (see XPath)
XML pointer language (Xpointer), 20
XML Schema, 871–896
advantages over DTDs, 10
built-in datatypes, 46
datatypes, 552, 875–890
elements and attributes, 871–875
standards, 18
stylesheets, 890–896
support for, 4
XSLT 2.0, 419, 926
xml:base attribute, 253
xml:lang attribute, 49
xml:space attribute, 49, 142, 498, 518
xmlns:xsl attribute, 520
XPath (XML Path Language), 45–112, 545–562
anchors, 903–904
attribute value templates, 66
axes, 548
boolean operators, 82
comments in expressions, 102
context, 550
data model, 46–54
datatypes, 67–71, 551–555

document root elements versus XML root elements, 105
keywords, 556–561
location paths, 55–66
node tests, 547
node types, 545
operator precedence, 562
operators, 71–102, 556–561
reluctant quantifiers, 900–902
slash (/) operator, 77
standards, 16
view of XML documents, 104–112
XSLT 2.0 processors, 104
XPath 1.0
comparing sets of nodes, 97
context, 55, 934
datatypes, 67, 551
dynamic typing, 73
modulo (mod) operator, 80
operator precedence, 562
XPath 2.0
addition, 73
atomic values, 83
collection() function, 271
comments in expressions, 102
compare() function, 291
constructor functions, 92
context, 56, 934
current node list, 935
datatype operators, 93–97
datatypes, 68, 70, 552
distinct-values() function, 218
division, 78
escaping single quotes, 174
every and some operators, 89
index-of() function, 263
integer division, 79
migration features, 925
modulo (mod) operator, 80
multiplication, 77
node operators, 100–102
node tests, 51
node type versus role, 105
operator precedence, 562
operators, 88–102
regular expression language, 903
replace() function, 173
sequences and atomic values, 52
set operators, 97–100
standards, 16
subtraction, 75
wildcards for namespace prefixes, 62
XPath 2.0 Data Model (XDM), 17
XPath expressions
attributes containing, 195
creating lookup tables, 256
xpath-default-namespace attribute, 362, 520
Xpointer (XML pointer language), 20
XQuery
integration with, 4
XQuery 1.0
compare() function, 291
datatypes, 70
index-of() function, 263
operator precedence, 562
standards, 17
xs:anyAtomicType atomic value type, 552
xs:anyAtomicType datatype, 71
xs:anySimpleType datatype, 70
xs:anyType datatype, 70
xs:anyURI atomic value type, 552
xs:anyURI datatype, 70
xs:base64Binary atomic value type, 553
xs:base64Binary datatype, 70
xs:boolean atomic value type, 553
xs:boolean datatype, 68
xs:date atomic value type, 553
xs:date datatype, 69
xs:dateTime atomic value type, 553
xs:dateTime datatype, 69
xs:dayTimeDuration atomic value type, 553
xs:dayTimeDuration datatype, 70
xs:decimal atomic value type, 553
xs:decimal datatype, 69
xs:double atomic value type, 553
xs:double datatype, 69
xs:duration atomic value type, 554
xs:duration datatype, 69
xs:float atomic value type, 554
xs:gDay atomic value type, 554
xs:gMonthA atomic value type, 554
xs:gMonthDay atomic value type, 554
xs:gMonthDay datatype, 70
xs:gYear atomic value type, 554
xs:gYearMonth atomic value type, 554
xs:gYearMonth datatype, 70
xs:hexBinary atomic value type, 555
xs:hexBinary datatype, 70

xs:integer datatype, 69
xs:NOTATION atomic value type, 555
xs:QName atomic value type, 555
xs:QName datatype, 70
xs:string atomic value type, 555
xs:string datatype, 68
xs:time atomic value type, 555
xs:time datatype, 69
xs:untyped atomic value type, 555
xs:untyped datatype, 71
xs:untypedAtomic atomic value type, 555
xs:untypedAtomic datatype, 71
xs:yearMonthDuration atomic value type, 555
xs:yearMonthDuration datatype, 70
XSL (Extensible Stylesheet Language)
  standards, 18
xsl namespace prefix, 11
<xsl:analyze-string> element, 362–368
<xsl:apply-imports> element, 368–371
<xsl:apply-templates> element, 151, 210, 214, 218, 238, 372–377
<xsl:attribute> element, 377–383
<xsl:attribute-set> element, 383–387
<xsl:call-template> element, 151, 387–390, 926
<xsl:character-map> element, 391–395
<xsl:choose> element, 88, 148, 263, 395–398
<xsl:comment> element, 397–401
<xsl:copy> element, 132–139, 401–405
<xsl:copy-of> element, 132–139, 405–408
<xsl:decimal-format> element, 127–130, 408–412, 920
<xsl:document> element, 412–416
<xsl:element> element, 416–422
<xsl:fallback> element, 278, 281, 282, 422–424
<xsl:for-each> element, 149, 207, 214, 227, 424–426
<xsl:for-each-group> element, 214, 228, 229, 260, 426–434
<xsl:function> element, 262, 279–281, 435–438
<xsl:if> element, 146, 263, 439
<xsl:import> element, 33, 440–443
<xsl:import-schema> element, 890–892
<xsl:import-schema> element, 443–445
<xsl:include> element, 33, 446–448
<xsl:key> element, 33, 194–198, 224, 448–451
<xsl:matching-substring> element, 451
<xsl:message> element, 453–455
<xsl:namespace> element, 455–457
<xsl:namespace-alias> element, 457–460
<xsl:next-match> element, 460–463
<xsl:non-matching-substring> element, 463
<xsl:number> element, 118–127, 131, 465–477, 920
<xsl:otherwise> element, 148, 263, 477–480
<xsl:output> element, 30, 480–487
<xsl:output-character> element, 487–488
<xsl:param> element, 33, 152–167, 489–493
<xsl:perform-sort> element, 214, 215–219, 494–498
<xsl:preserve-space> element, 33, 141–142, 498–500
<xsl:processing-instruction> element, 500–502
<xsl:result-document> element, 278, 502–507
<xsl:sequence> element, 53, 216, 508–511
<xsl:sort> element, 205–215, 227, 254, 511–517
<xsl:strip-space> element, 33, 141–142, 517–519
<xsl:stylesheet> element, 256, 519–523
<xsl:template> element, 28, 31, 152, 524–525
<xsl:text> element, 114, 526–528
<xsl:transform> element, 528
<xsl:value-of> element, 115, 116–118, 528–531
<xsl:variable> element, 33, 162, 167, 222, 531–536
<xsl:when> element, 148, 263, 536–538
<xsl:with-param> element, 152–167, 539–544
XslCompiledTransform object, 298
XSLT (Extensible Stylesheet Transformations)
  history, 1–3
XSLT 1.0
  addition, 72
  conditional expressions, 88
  match and use attributes, 195
  output methods, 480
  parameter passing, 163
  standards, 16
  undefined parameters, 159
  variables at same level with same name, 533
  <xsl:value-of> element, 529

XSLT processors, xii
XSLT 1.0 processors
common extension elements, 507
forwards-compatible mode, 364
XSLT 2.0
addition, 72
browser support, xiii
collation attribute, 195
combining XML documents, 260–268
converting to boolean values, 146
creating new functions, 279–281
custom collations, 287–293
data-type attribute, 213
datatypes, 12
design requirements, 3
dividing <xs:double> elements by zero, 412
doc() and doc-available() function, 269–271
elements attribute, 498, 518
escaping single quotes, 174
EXSLT functions, 331
formatting dates and times, 130–132
formatting functions, 130
grouping, 228–242
idref() function, 188
migration features, 925
mode attribute, 159, 376, 524
new features, 515
output methods, 480
parameters, 158–167
passing sequences of generate-id() functions, 203
priority attribute, 525
processors, 104
rules for passing parameters, 390
schema features, 419
self axis, 64
separator attribute, 381
sequences, 89
specifying first elements using predicate expressions, 201
standards, 16
subtraction, 75
to operator, 149
type checking, 928
unparsed-text() and unparsed-text-available() functions, 272
variables containing nodes, 536
xpath-default-namespace attribute, 522
<xsl:number> element, 126–127
<xsl:perform-sort> element, 215–219
<xsl:result-document> element, 278
<xsl:sort> element, 214
<xsl:text> element, 526
<xsl:value-of> element, 116–118, 529
XSLT processors, xii
XSLT 2.0 processors
avoiding recursion, 250
XSLT extensions
writing in other languages, 326–330
XSLT processors, xviii
(see also Altova XSLT engine; .NET framework; Saxon XSLT processor; Xalan XSLT processor; XSLT 1.0 processors; XSLT 2.0 processors)
EXSLT support, 331
installing, 20–24
XSLT stylesheets (see stylesheets)
XsltArgumentList object, 157

Y

year-from-dateTime() function, 865
year-from-date() function, 864–865
years-from-duration() function, 866–868

Z

Z character groups, 911
zero-digit attribute, 409
zero-or-one() function, 868–870
Zl character group, 911
Zp character group, 911
Zs character group, 911

About the Author

Doug Tidwell is a senior programmer at IBM. He has more than a sixth of a century of programming experience, and has been working with markup languages for more than a decade. He was a speaker at the first XML conference in 1997 and has taught XML classes around the world. His job as a cyber evangelist is to look busy and to help people use new technologies to solve problems. Using a pair of zircon-encrusted tweezers, he holds a master's degree in computer science from Vanderbilt University and a bachelor's degree in English from the University of Georgia. He lives in Raleigh, North Carolina, with his wife and their daughter Lily.

Colophon

The animal on the cover of *XSLT*, Second Edition, is a Jabiru (*Jabiru mycteria*). Standing up to five feet tall and with a wingspan of eight feet, this wading stork is the largest flying bird in the western hemisphere. The bird's habitat ranges from southern Mexico to northern Argentina, and much of its migrating population is found in Belize from November through July. Its habitat generally includes coastal areas, savannas, and marshes, and it feeds on freshwater wildlife such as frogs, fish, and even snakes. Its plumage is mostly white, but its head, neck, and beak are jet black. A featherless red pouch at the base of its neck gives the Jabiru its name, which means "swollen neck" in the Tupi-Guarani language.

The Jabiru population has steadily decreased over the past decades due to hunting and deforestation, but some areas of Central America have seen a recovery in the bird's numbers. It is currently considered a species of least concern by the World Conservation Union, an improvement from a status of near-threatened in 1988.

The cover image is an original antique engraving from the 19th century. The cover font is Adobe's ITC Garamond. The text font is Linotype Birka, the heading font is Adobe Myriad Condensed, and the code font is LucasFont's TheSansMonoCondensed.